Designing Efficient and Scalable Key-value Cache Management Systems

Juncheng Yang

CMU-CS-24-149
December 2024

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

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

The work presented in this thesis was sponsored by the National Science Foundation under award numbers CCF1910813, CNS1901410, CNS1956271, and CCF1943409 and a Facebook Fellowship. The views and conclusions in this document are those of the author and should not be interpreted as representing the official policies of any sponsoring institution, the U.S. government, or any other entity.

Keywords: Storage System, Efficiency, Scalability, Cache Management, Cache Replacement, Key-Value Cache, Segment-structured

"Learn something about everything and everything about something."

–Thomas Huxley

Abstract

Software caches have been widely deployed at scale in today's computing infrastructure to improve data access latency and throughput. These caches consume PBs of DRAM across the industry, necessitating high efficiency — reducing DRAM consumption without compromising the miss ratio. Meanwhile, modern servers have hundreds of cores per CPU, making thread scalability a critical requirement for designing software caches. This thesis explores multiple approaches from system and algorithm perspectives to improve the efficiency and scalability of software caches.

This thesis has two parts. The first part focuses on new *system designs* that allow caches to store *more objects* in the cache to achieve a low miss ratio. In this part, I will describe three works. First, I will discuss a large-scale production key-value cache workload analysis. Second, drawing on insights from the workload study, I will describe the design of Segcache, a TTL-indexed segment-structured key-value cache that removes expired objects quickly, uses tiny object metadata, and enables close-to-linear scalability. Third, I will present C2DN and demonstrate how to use erasure coding to design a highly efficient fault-tolerant CDN cache cluster.

The second part focuses on new *algorithms* that allow the cache to store *more useful objects* in the cache, which is also critical for cache efficiency. In this part, I will discuss four works. First, I will investigate the design of a low-overhead learned cache. Existing caches that use machine learning often incur significant storage and computation overheads. I will show a new approach — learning on the group level, which amortizes overheads and accumulates more information for better learning. While GL-Cache is faster than existing learned caches, it is still more complex compared to state-of-the-art heuristics. In the following chapter, I will discuss two foundational techniques, lazy promotion and quick demotion, which enable us to design simple yet effective eviction algorithms. In the third chapter of this part, I will demonstrate an example using the two techniques, S3-FIFO, a new eviction algorithm only composed of FIFO queues but achieves higher efficiency and scalability than 11 state-of-the-art algorithms. In the last chapter of this part, I will introduce SIEVE, a new eviction algorithm that uses a single queue to achieve lazy promotion and quick demotion. SIEVE is simpler than LRU, but achieves state-of-the-art efficiency and scalability.

This thesis will demonstrate how we leverage large-scale measurements to obtain insights for new system and algorithm designs, which allow modern software caches to achieve high efficiency and close-to-linear scalability. Several of the designs, i.e., Segcache, S3-FIFO, and SIEVE, have made into real world deployment. Moreover, the artifacts open-sourced as part of this thesis, i.e., libCacheSim and cache traces have been widely used by the community.

Acknowledgments

I would like to start by thanking my advisor, Rashmi Vinayak, who provided extensive support and guidance throughout my Ph.D. journey. I remember that in my first semester, I had no idea how to do research. I learned a lot from Rashmi on how to think about research, what I should focus on, and how to write a good paper. Besides all the mentoring and guidance, I am also truly grateful for the trust and freedom she bestowed on me. The trust and freedom allowed me to explore many different directions and enabled many of the new exciting discoveries in this thesis. In addition to supporting my research, Rashmi is also a great mentor and has spent tremendous effort helping me improve my technical speaking and writing skills. Words cannot really express my gratitude, and I appreciate all the help, guidance, and support from my advisor.

In addition to my advisor, I want to express deep gratitude to members of Parallel Data Lab and TheSys group, such as Greg Ganger, Phillip Gibbons, and many others, with whom I had many technical discussions and who inspired many new ideas.

A big component of a Ph.D. is to learn from many different people; I am grateful for my awesome collaborators, especially Yao Yue, who shared extensive production experience and insights with me. The lessons that she taught me are invaluable, and I cannot learn elsewhere without years of effort. Besides Yao, several other collaborators, including Daniel Berger and Ramesh Sitaraman, also taught me much about research when I started my Ph.D. journey. They provided priceless insights that helped me become a better researcher. From these amazing collaborators, I learned how to choose a problem to work on. They helped shape this thesis and my research taste.

I would also like to extend my appreciation to the undergraduate and graduate students who I have had the pleasure to work with over the years. They taught me a lot about communication, collaboration, and mentorship. There is a long list of people I would like to acknowledge, including, but not limited to Jack Kosaian, Saurabh Kadekodi, Francisco Maturana, Ziyue Qiu, Yazhuo Zhang, Daiyaan Arfeen, Yixuan Mei, Ziming Mao, and Qinhan Chen.

As a student, I have to deal with administrative tasks, e.g., travel and organizing events, which can take a tremendous amount of time. I would like to give a big shout-out to Deb and Matt from the Computer Science Department and Karen and Joan from PDL, who have made my life much easier. They are truly amazing and the best that I have worked with.

Last but not least, I would like to acknowledge my family for their unwavering support throughout my journey. My wife, Jia Song, and my parents have stood behind me unconditionally. I could not have accomplished this without them.

Contents

List of Figures

List of Tables

Part I

Setting the Stage

Chapter 1

Introduction

The widespread adoption of personal computing devices, such as laptops and mobile phones, in the last twenty years has given rise to a multitude of online services, e.g., social networking, gaming, and shopping. Data is a critical component to support these online activities. However, data are often stored on bulk storage devices, such as hard disk drives (spinning disks). Accessing data on spinning disks is very slow, in the order of 10s of milliseconds or longer. Software caches are widely used to bridge the gap between fast compute and slow storage. The main idea is to store frequently accessed data on a fast storage device so that future accesses to these data can be served quickly. There are different types of caches, e.g., hardware caches and software caches. In contrast to hardware caches, the management of software caches, e.g., eviction, is implemented in software.

1.1 The Ubiquitous Deployment of Software Caches

Software caches are widely deployed across system stacks in today's data centers and personal devices. For example, block caches, e.g., Linux page cache [206], reduce data access latency; key-value caches, e.g., Memcached [228], and Cachelib [95], are often used to reduce repeated computation and improve application scalability; and object caches, such as CDNs, reduce data access latency and bandwidth costs. Large-scale cache deployments have become the foundation for supporting our digital society. However, these deployments often consume a huge amount of resources, both storage and computation. For example, Twitter reports that 100s of TB of DRAM and 100,000s of CPU cores are used for caching in 2020 [370]; Pinterest's Memcached fleet consumes 460 TB of DRAM spanning over 5000 EC2 instances and 70 cache clusters in 2022 [204]; Netflix caches around 14 PB of application data (not including the movies) using 18,000 servers in 2022 [255].

The performance of a cache is often measured using two metrics: *efficiency* and *throughput*. Efficiency measures how many requests can be fulfilled by the cache and is often assessed using miss ratio — the fraction of requests that are cache misses. Throughput is often reported using the number of requests (including both hits and misses) the cache can serve per second (MQPS). It is an indicator of the computation resources needed to serve one request. Besides single-threaded throughput, multi-threaded throughput has become increasingly critical because modern CPUs possess an abundance of cores; for example, a high-end AMD server CPU in 2022 features 192 hyper-threads [11].

Even a small reduction in resource (computation and storage) consumption not only leads to substantial cost savings but also enhances the sustainability of data serving. For example, a 10% cache size reduction for Twitter translates to a reduction of tens of TB of DRAM or thousands of servers. Meanwhile, unlike batch computation jobs (e.g., machine learning training jobs) that can be time-shifted or location-shifted to use less carbon-intensive energy, stateful services such as caches must always be online and close to the user (e.g., computation jobs or end-users). As a result, improving resource efficiency is by far the only solution to reduce both embodied carbon footprint and operational carbon footprint.

For the aforementioned reasons, researchers have put tremendous effort into improving software cache efficiency over the past few decades. However, the majority of these works focus on the eviction algorithm [38, 42, 43, 109, 165, 172, 227, 291, 312, 329, 397]. Besides improving cache efficiency, an increasing number of works looked into the throughput and scalability of a cache in recent years [118, 203, 276]. These works often trade off cache efficiency for improved throughput and scalability.

1.2 Overview

This thesis takes a holistic view of a cache and explores multiple approaches from both system and algorithm perspectives to improve the efficiency and scalability of a software cache. Leveraging observations and insights from production systems, this thesis demonstrates (1) how to use object grouping, metadata sharing, and proactive expiration to improve space utilization and reduce the cache miss ratio; (2) how erasure coding interacts with caching when designing a fault-tolerant cache cluster and demonstrates a novel way to use erasure coding for caching; and (3) how to make use of machine learning for cache evictions practical; and (4) how to design simple, scalable, and efficient software cache eviction algorithms using two new techniques, lazy promotion and quick demotion. This thesis has two parts: the first focuses on the new key-value cache system designs, while the second focuses on the new cache eviction algorithm designs. The remainder of this thesis is outlined as follows.

Chapter 2 provides background information on different types of software caches and how performance is measured. Then it introduces a new concept called "key-value cache management system". A key-value cache management system has two components, "cache replacement" and "space management". Cache replacement decides which objects to store in the cache, and space management decides how objects are stored and looked up in the cache. Both components play crucial roles in a cache's performance. A better cache replacement keeps more useful objects in the cache, and better space management incurs less overhead, e.g., metadata and fragmentation.

Chapter 3 describes a large-scale analysis of production in-memory key-value cache workloads at Twitter. Several production workload analyses have fueled research in improving the effectiveness of in-memory caching systems [25, 259]. However, the coverage is still sparse, considering the wide spectrum of cache use cases. This chapter shows the analysis of 153 in-memory key-value cache clusters at Twitter. It characterizes cache workloads based on traffic pattern, time-to-live (TTL), popularity distribution, and object size distribution. A fine-grained view of different workloads uncovers the diversity of use cases: many are more write-heavy or skewed than previously shown, and some display unique temporal patterns. This work was published at OSDI'20 [370] and invited to submit to TOS'21 [371].

Inspired by the workload study showing the prevalence of small objects and extensive use of TTLs, **Chapter 4** describes a new cache storage design, Segcache. Segcache uses a segment-structured (similar to log-structured storage [264, 292]) design that stores data in fixed-size segments with three key features: (1) it groups objects with similar creation and expiration time into the segments for efficient expiration, (2) it approximates some and lifts most per-object metadata for object metadata sharing and reduction, and (3) it performs segment-level bulk expiration and eviction with fewer and smaller critical sections for high scalability. Evaluation using production traces from Twitter shows that Segcache uses 22-60% less memory than state-of-the-art designs for various workloads. Segcache simultaneously delivers high throughput under single and multiple threads: up to 40% higher and 8× than Memcached on 1 and 24 thread(s), respectively. Segcache improves the space utilization of a key-value cache by reducing the space wasted on fragmentation, object metadata, and expired objects, all of which contribute to Segcache's high efficiency. This work was published at NSDI'21 [372], won a best-paper award, and has been in production at Twitter and Momento.

Chapter 5 describes the design of an efficient fault-tolerant cache cluster. Content Delivery Networks (CDNs) deploy cache clusters around the globe so that data can be delivered to end users both quickly and cheaply. However, servers in edge clusters often suffer from unavailability. To mitigate the impact (miss ratio spike) from server unavailability, CDN operators, e.g., Akamai, often add redundancy to the edge cluster using replication. This chapter explores how to leverage erasure coding to reduce the storage overhead of redundancy. This chapter shows that direct use of erasure coding in CDN caches is insufficient due to write load imbalance — erasure-coded chunks are evicted at different times from the servers. Therefore, it shows a new design, coded content delivery network (C2DN), which leverages parity chunks to perform distributed load balancing so that chunks of the same object are evicted at a similar time. Implemented on top of open-source production software, this chapter demonstrates that C2DN obtains an 11% lower byte miss ratio than replication-based CDNs, eliminates unavailability-induced miss ratio spikes, and reduces write load imbalance by 99%. This work was published at NSDI'22 [373].

Chapter 6 describes the exploration for efficient eviction algorithms for segment/log-structured caches. Several recent works have looked into machine-learning-assisted cache eviction algorithms, which we call "learned caches". This chapter categorizes existing works into three types of learned caches: object-level learning, learning from distribution, and learning from simple experts. The learning granularity in existing approaches is either too fine (object-level), incurring significant computation and storage overheads, or too coarse (workload or expert-level) to capture the differences between objects, leaving a considerable efficiency gap. In this chapter, we introduce a new approach to using machine learning in caches, which clusters similar objects into groups and performs learning and eviction at the group level, which we call "group-level learning". Learning at the group level accumulates more signals for learning, leverages more features with adaptive weights, and amortizes overheads over objects, achieving high efficiency and throughput. We implemented group-level learning on an open-source production cache, which we call GL-Cache, a group-level learned cache. Group-level learning naturally maps to a segment-structured cache where each group is a segment. Moreover, the idea generalizes beyond Segcache and can be used for any cache storage design. Evaluations on 118 production block I/O and CDN cache traces show that GL-Cache has a higher hit ratio and higher throughput than state-of-the-art designs. This work was published at FAST'23 [374].

LRU has been the basis of cache eviction algorithms for decades, with a plethora of innovations on improving LRU's miss ratio and throughput. While it is well-known that FIFO-based eviction algorithms provide significantly better throughput and scalability, they lag behind LRU on miss ratio, thus, cache efficiency. **Chapter 7** describes a large-scale measurement of cache eviction algorithms using huge datasets of 6587 block and web cache workloads collected in the past two decades. This chapter shows a new finding that, contrary to common wisdom, some FIFO-based algorithms, such as FIFO-Reinsertion (or CLOCK), are, in fact, more efficient (have a lower miss ratio) than LRU. Moreover, it demonstrates that QUICK DEMOTION — evicting most new objects very quickly — is critical for cache efficiency. Moreover, this chapter illustrates that when enhanced by QUICK DEMOTION, not only can state-of-the-art algorithms be more efficient, a simple FIFO-based algorithm can outperform five complex state-of-the-art in terms of miss ratio.

Built on the lessons in **Chapter 7**, the next chapter describes a new cache eviction algorithm that is simple yet effective. **Chapter 8** describes a new **s**imple and **s**calable FIFO-based algorithm with three **s**tatic queues (S3-FIFO). Evaluated on 6594 cache traces from 14 datasets, S3-FIFO has lower miss ratios than state-of-the-art algorithms across the diverse datasets. Moreover, S3-FIFO's efficiency is robust — it has the lowest mean miss ratio on 10 of the 14 datasets and is among the top algorithms on the other datasets.

The use of FIFO queues enables S3-FIFO to achieve good scalability with 6× higher throughput compared to optimized LRU at 16 threads. The insight is that most objects in the cache workloads will only be accessed once in a short window, so it is critical to evict them early. The key of S3-FIFO is a small FIFO queue that filters out most objects from entering the main cache. Moreover, this chapter shows that filtering with a small static FIFO queue has a guaranteed eviction time and higher eviction precision compared to state-of-the-art adaptive algorithms.

While S3-FIFO is much simpler than state-of-the-art eviction algorithms, it still incurs non-trivial implementation complexity compared to simple heuristics such as FIFO and LRU. Taking the insights of LAZY PROMOTION and QUICK DEMOTION in **Chapter 7** further, Chapter 9 describes a new cache eviction algorithm, SIEVE, that is simpler than LRU, while achieving state-of-the-art efficiency and scalability on web cache workloads. SIEVE can be implemented in multiple production cache libraries with 20 lines of code. Meanwhile, it achieves up to 63.2% lower miss ratio than state-of-the-art eviction algorithms such as ARC. Besides being an eviction algorithm, the simplicity also allows SIEVE to be a cache primitive that can be used to design more advanced eviction algorithms. We show that simply replacing LRU in 2Q, ARC, and LeCaR would further boost the efficiency of these algorithms. This work was published at NSDI'24 [385].

1.3 Dissertation Contributions and Impacts

Caching research has a long history with many efforts devoted to designing more efficient eviction algorithms [38, 105, 109, 165, 172, 227, 312, 329]. This thesis takes a holistic view of software caching and explores different perspectives on improving the efficiency of caching systems. Specifically, the investigations in this thesis were driven by (1) **new workload patterns**, such as the dominance of small objects and the wide use of TTLs (Chapter 3 and Chapter 4); (2) **new hardware trends** such as the increasing number of cores per CPU and the asymmetric read and write performance in storage devices (Chapter 4); (3) **new requirements**, such as the need for fault tolerance in Content Delivery Network caches (Chapter 5); and (4) **new observations**, e.g., FIFO with reinsertion is more efficient than LRU; one-hit wonders are prevalent in cache workloads, and thus quickly removing them is important (Chapter 7 and Chapter 8).

This thesis makes the following contributions.

- It describes a large-scale key-value cache workloads study to illustrate how key-value caches are used in production systems. The study highlights the diversity of workloads, including write-heavy and skewed patterns, as well as the significance of TTL in shaping cache behavior. These findings offer insights for optimizing cache performance and resource utilization.
- To address the challenges identified in the workload analysis, this thesis presents a new storage layout called Segcache. For the first time, an approximate TTL index is introduced to the key-value cache to improve efficiency. The new index enables efficient expiration and eviction of data by grouping objects with similar lifespans into segments. By approximating and sharing object metadata, the design significantly reduces storage overhead. Additionally, it performs segment-level bulk operations to minimize critical section size, thus contributing to high scalability and performance.
- Recognizing the importance of reliability in Content Delivery Networks, this thesis demonstrates the first fault-tolerant CDN cluster that delivers low miss ratio, high availability, and near-perfect write load balancing. The core contribution lies in introducing parity rebalance in erasure coding, which enables efficient cache synchronization across servers.
- While machine learning has shown promise in enhancing cache efficiency, existing methods suffer from high overheads. To mitigate this, this thesis showcases a group-level learning approach. By grouping similar objects and training models on these groups, the new design can reduce storage and computational costs while achieving high cache efficiency.
- This thesis describes the largest-scale eviction algorithm measurement study, which revealed that lazy promotion and quick demotion are critical for cache performance and efficiency. This thesis then presents the first FIFO-queue-only eviction algorithm that leverages this insight. S3-FIFO is efficient and scalable. Meanwhile, it is simpler than state-of-the-art cache eviction algorithms.
- To further simplify the design of cache eviction algorithm, this thesis presents SIEVE, an eviction algorithm that performs lazy promotion and quick demotion using only one queue. SIEVE achieves state-of-the-art efficiency and scalability while being simpler than LRU.

This thesis introduces a new concept: **cache management system**. A cache management system has two components: *cache replacement* and *space management*. A good cache replacement allows the cache to store more useful objects, and good space management allows the cache to store more objects. This thesis demonstrates that both cache replacement and space management are important for cache efficiency and scalability. Leveraging insights from large-scale measurements, we demonstrate several approaches to boost the efficiency and performance of cache management systems.

Moreover, the work in this thesis has impacted both industry and academia.

Adoption in the industry. Several works presented in this thesis have been adopted for production. For example, Segcache has been adopted for production at Twitter and Momento; Variants of S3-FIFO have been adopted for production by Google, VMware, and Redpanda, among others; and SIEVE has been adopted by Meta and many startups.

Adoption in the open-source community. The open-source community has embraced these innovations, with Segcache re-implemented and open-sourced by Twitter, S3-FIFO integrated into over 20 open-source libraries and systems, and SIEVE available in even more cache libraries across 16 programming languages.

Educational Influence. Beyond practical applications, this thesis has also influenced educational and research communities. Several works, including workload analysis, S3-FIFO, and SIEVE, are now parts of various courses, blogs, meetups, and reading groups.

Research artifacts. Additionally, as part of the works in this thesis, we have released research artifacts to the public domain. They include (1) production cache traces [1], which have been used in hundreds of subsequent research studies; and (2) a new high-performance cache simulator, libCacheSim [2], which has been adopted by nearly 100 institutions and companies worldwide.

[1]The traces can be downloaded at `https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/cacheDatasets`.

[2]`https://libcachesim.com`.

Chapter 2

Background and related work

2.1 Software caches

While sometimes less noticed, software caches are pervasively deployed across system stacks in both data centers and personal devices. Figure 2.1 shows a typical web service architecture with each box representing one service. In the diagram, caches appear in almost every box. For example, distributed key-value cache clusters, page caches in the storage servers and personal devices, transient hot object caches in application services, and CDN caches deployed close to users.

The omnipresent software caches can have different interfaces. For example, distributed key-value caches often use the Memcached interface, storage caches often use the block interface, and CDN caches all use the HTTP interface.

Besides the interface, different caches may also use different storage mediums, e.g., DRAM and flash. Designing caches for some storage mediums may require meeting extra metrics. For instance, flash cache designs often have requirements on write amplification and flash endurance due to limited flash lifetime [225, 320]. The works in this thesis mostly focus on DRAM caches (except C2DN in Chapter 5) and thus only focus on efficiency (miss ratio) and throughput scalability.

Along with the differences in interface and storage medium, production caches also differ in the types of deployments. For example, many application caches, e.g., Facebook's social graph [54, 259] cache, are in-process caches using an embedded library. Standalone caches, like Memcached, often use distributed deployment with consistent hashing for

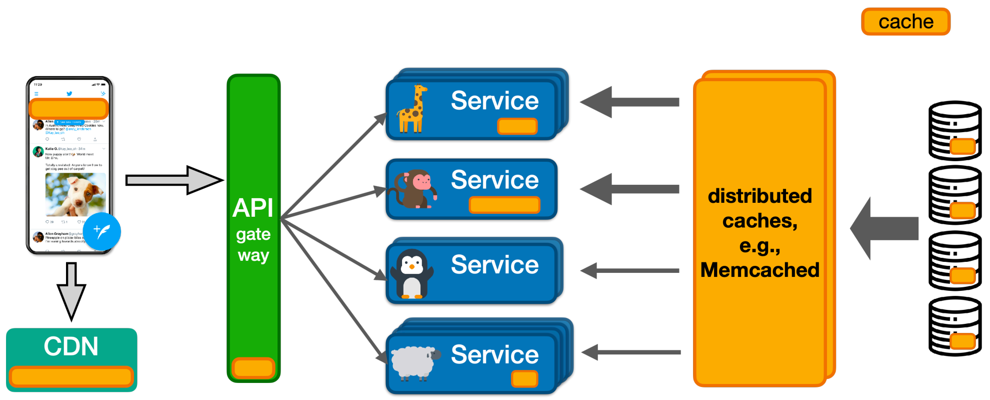


Figure 2.1: Software caches are widely deployed in today's data centers.

partitioning.

Although bearing many differences in the interface, storage medium, and deployment type, the core functions and features of software caches are similar — storing a subset of frequently requested data in a limited space to minimize the number of cache misses.

2.2 Metrics of a software cache

We use two metrics to measure a cache's performance: miss ratio and throughput.

Efficiency. Miss ratio measures the efficiency of a cache. A lower miss ratio means more requests can be served directly from the cache, translating to faster data access and lower cost.

Throughput. Throughput measures the number of requests the cache can serve per second (QPS), which includes both cache hits and cache inserts (from misses). Because the goal of a cache is to serve data faster, it is critical for a cache to have a high throughput. Besides the throughput in a single thread, the ability to use multiple cores effectively — cache scalability is also very important because today's server CPUs have many CPU cores, e.g., a high-end AMD server CPU can have 200 cores [11].

2.3 Key-Value cache management system

For CPU caches and block storage caches, the eviction (and prefetch) algorithm is the only factor that decides the efficiency. However, distributed key-value caches are more complex, with multiple components contributing to the efficiency. We propose the concept of a “key-value cache management system” that has two components: cache replacement and space management (Figure 2.2). Cache replacement decides which object should be stored in the cache, while space management decides how objects are stored and looked up in the cache. Cache replacement decides the replacement effectiveness, and space management decides the space utilization. Both components play critical roles in the efficiency of a cache. A cache with better replacement makes more clever decisions on what objects to store in the cache so that it can have a lower miss ratio. A cache with better space management wastes less space on fragmentation, metadata, and redundancy, which allows it to cache more objects in the limited space and achieve a lower miss ratio.

2.3.1 Cache replacement

Cache replacement has three pieces: eviction, admission, and expiration.

Eviction. The eviction algorithm is the most important component of cache replacement and has been widely studied. For example, Greedy-Dual variants [64, 193], LFU variants [22, 105], MultiQueue [397], LIRS [165], ARC [227], 2Q [172], TinyLFU [108, 109], LHD [38], Hyperbolic [48], LRB [312], LeCaR[329] and CACHEUS [291] among many other eviction algorithms have been designed to improve cache efficiency for different workloads.

Admission. While the eviction algorithm has always been critical for cache efficiency, cache admission has gained increasing popularity in recent years [27, 43, 202, 219, 348]. Cache admission decides whether to store an object during a cache miss. Because KV-cache workloads often follow Zipf distribution (Chapter 3) with many objects receiving one or very few requests (long tail), rejecting unpopular objects from entering the cache can reduce cache pollution and miss ratio. Moreover, cache admission also reduces wearout and tail latency for flash caches.

Expiration. Time-to-live (TTL) is widely used in KV-cache workloads (Chapter 3). Therefore, timely removing expired objects becomes important for cache efficiency. Unlike

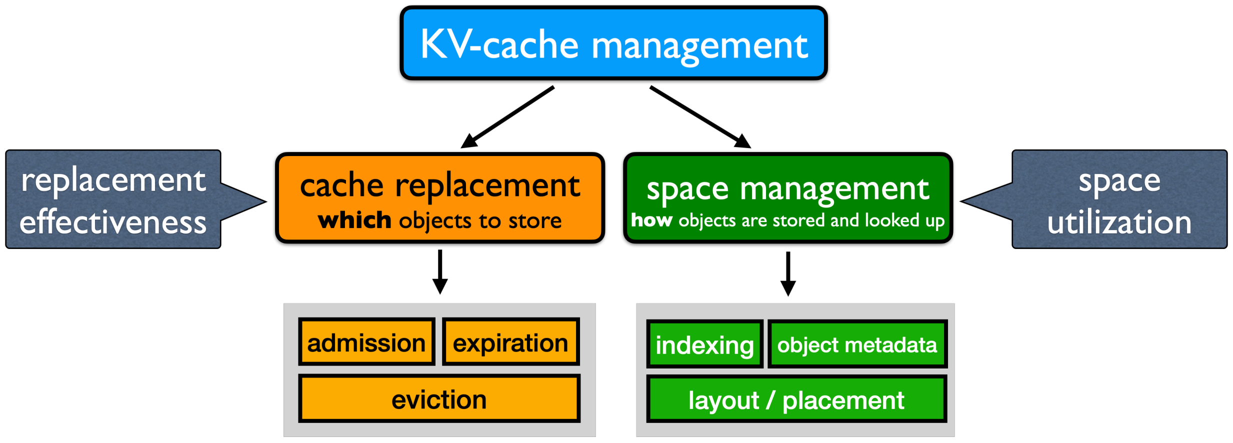


Figure 2.2: A KV-cache management system consists of two components: cache replacement and space management, both of which affect cache efficiency.

eviction, which removes objects that may be requested in the future, expired objects cannot be used and should be removed as quickly as possible. Adding an index on TTL or expiration time limits scalability and increases object metadata size. Therefore, production systems use different approaches. For example, Memcached uses scanning, and Redis uses sampling to find and remove expired objects. However, scanning and sampling consume a non-trivial amount of CPU cycles and provide no guarantee on how long expired objects may overstay in the cache.

2.3.2 Space management

Variable-sized objects and small objects introduce three challenges to KV-cache efficiency. First, variable-sized memory allocations lead to fragmentation [293]. Second, object metadata wastes precious cache space when objects are small. Third, the index for lookup may consume too much DRAM for tiny objects. Therefore, space management in a key-value management system includes three ingredients: storage layout, indexing structure, and object metadata.

Storage layout Storage layout decides how objects are placed on the storage device. External management, such as malloc and file systems, are common choices, especially for systems not dedicated to caching (caching is one of the many features). However, external management often suffers from fragmentation problems [293], as well as occasional long tail latency [273]. For flash cache, it further introduces tail latency and endurance problems [320].

Slab storage organizes the heap based on object size class and is the most common option for in-memory caches such as Memcached and Cachelib. Slab storage divides space into fixed-sized slabs, and each slab is further carved into equal-sized chunks to store data. The chunks in one slab have the same size, but different slabs can have different chunk sizes. Compared to using “malloc”, slab storage does not incur external fragmentation, thus, having a bounded memory usage — an important property for reducing memory over-provisioning in production. However, slab storage has internal fragmentation where space is wasted when an object is smaller than the chunk holding it. Besides, slab storage can only evict objects with sizes similar to the incoming object and often suffers from slab calcification: when object size distribution changes over time, there are insufficient slabs with the high-demand chunk size. Rebalancing slabs (evicting all objects from a slab and changing the chunk size) may solve the calcification problem, but when and how to rebalance remains a challenging research problem.

To reduce fragmentation, log-structured storage is employed in some KV-cache systems [14, 203]. However, log storage limits the eviction algorithm to FIFO variants, significantly reducing cache efficiency.

Indexing structure When a workload mostly consists of tiny objects (< 100B), the indexing structure (B-tree or hash table) may consume too much memory. Reducing an indexing structure’s memory consumption is critical for both DRAM caches and flash caches because (1) the huge DRAM requirement may limit the usable space of a flash cache, and (2) the index reduces the usable space for DRAM caches. A common solution is to use a set-associative cache, which combines the cache with the hash table. For example, Twitter designs an in-memory cuckoo cache to cache tiny objects, and Meta designs a set-associative flash cache for tiny objects in Cachelib. However, a set-associative cache causes conflict misses, increasing the miss ratio. In Chapter 4, we demonstrate a different approach to reducing the hash table’s memory consumption by sharing metadata.

Object metadata The second problem with caching tiny objects is the object metadata size. When objects are small, the per-object metadata becomes relatively large. For example, Memcached uses 56 bytes of metadata per object. However, many production caches have a mean object size of around 100 bytes (Chapter 3). The large per-object metadata wastes huge space. Therefore, reducing the per-object metadata without compromising features is important to improve a cache’s efficiency.

Part II

Efficient, Scalable and Reliable Key-value Cache

Chapter 3

Understanding Production Key-value Cache Workloads

Using in-depth measurements to examine how existing systems are used, the diverse patterns in the workloads, and the versatile requirements of the applications (users) is an important step before we can improve existing systems and design new systems. Key-value caches have been deployed in production systems for over two decades and have become a mature component of modern infrastructure. But how are they different from traditional caches, such as page caches? This chapter goes in-depth to explore the in-memory key-value cache workloads at Twitter.

3.1 Background

3.1.1 Service Architecture and Caching at Twitter

Twitter started its migration to a service-oriented architecture, also known as microservices, in 2011 [325]. Around the same time, Twitter started developing its container solution [15, 16] to support the impending wave of services. Fast forward to 2020, the real-time serving stack is mostly service-oriented, with hundreds of services running inside containers in production. As a core component of Twitter's infrastructure, in-memory caching has grown alongside this transition. Petabytes of DRAM and hundreds of thousands of cores are provisioned for caching clusters, which are containerized.

At Twitter, in-memory caching is a managed service, and new clusters are provisioned semi-automatically to be used as look-aside cache [259] upon request. There are two in-memory caching solutions deployed in production, Twemcache, a fork of Memcached[228], is a key-value cache providing high throughput and low latency. The other solution, named Nighthawk, is Redis-based and supports rich data structures and replication for data availability. In this work, we focus on Twemcache because it serves the majority of cache traffic.

Cache clusters at Twitter are considered *single-tenant*[1] based on the service team requesting them. This setup is very beneficial to workload analysis because it allows us to tag use cases, collect traces, and study the properties of workloads individually. A multi-tenant setup will make similar studies extremely difficult, as researchers have to tease out individual workloads from the mixture and somehow connect them to their use cases. In addition, smaller but distinct workloads can easily be overlooked or mischaracterized due to low traffic.

Unlike other cache cluster deployments, such as social graph caching [41, 54] or CDN caching [154, 347], Twemcache is mostly deployed as a single-layer cache, which allows us to analyze the requests directly from clients without being filtered by other caches. Previous work [154] has shown that layering has an impact on properties of caching workloads, such as popularity distribution. This single-tenant, single-layer design provides us the perfect opportunity to study the properties of the workloads.

3.1.2 Twemcache Provisioning

There are close to 200 Twemcache clusters in each data center as of 2020. Twemcache containers are highly homogeneous and typically small, and a single host can run many of them. The number of instances provisioned for each cache cluster is computed from user inputs including throughput, estimated dataset sizes, and fault tolerance. The number of instances of each cluster is automatically calculated first by identifying the correct bottleneck and then applying other constraints, such as the number of connections to support. The size of production cache clusters ranges from 20 to thousands of instances.

[1]Although each cluster is single-tenant, each tenant might cache multiple types of objects of different characteristics.

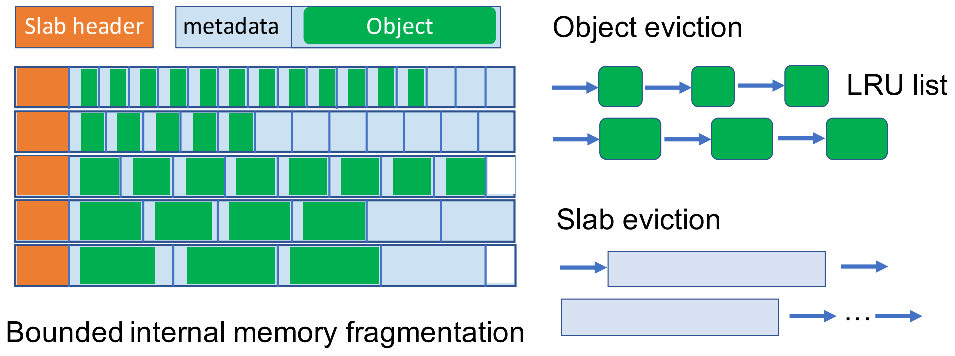


Figure 3.1: Slab-based memory management for bounded memory fragmentation. While Memcached uses object eviction, Twemcache uses slab eviction, which evicts all objects in one slab and returns the slab to a global pool.

3.1.3 Overview of Twemcache

Twemcache forked an earlier version of Memcached with some customized features. In this section, we briefly describe some of the key aspects of its designs.

Slab-based memory management Twemcache often stores small and variable-sized objects in the range of a few bytes to 10s of KB. On-demand heap memory allocators such as ptmalloc[135], jemalloc[161] can cause large and unbounded external memory fragmentation in such a scenario, which is highly undesirable in production environment, especially when using smaller containers. To avoid this, Twemcache inherits the slab-based memory management from Memcached (Figure 3.1). Memory is allocated as fixed size chunks called *slabs*, which default to 1 MB. Each slab is then evenly divided into smaller chunks called *items*. The *class* of each slab decides the size of its items. By default, Twemcache grows item size from a configurable minimum (default to 88 bytes) to just under a whole slab. The growth is typically exponential, controlled by a floating point number called growth factor (default to 1.25), though Twemcache also allows precise configuration of specific item sizes. Higher slab classes correspond to larger items. An object is mapped to the slab class that best fits it, including metadata. In Twemcache, this per-object metadata is 49 bytes. By default, a slab of class 12 has 891 items of 1176 bytes each, and each item stores up to 1127 bytes of key plus value. Slab-based allocator eliminates external memory fragmentation at the cost of bounded internal memory fragmentation.

Eviction in slab-based cache To store a new object, Twemcache first computes the slab class by object size. If there is a slab with at least one free item in this slab class, Twemcache uses the free item. Otherwise, Twemcache tries to allocate a new slab into this class. When memory is full, slab eviction is needed for allocation.

Some caching systems, such as Memcached, primarily perform item-level eviction, which happens in the same slab class as the new object. Memcached uses an approximate LRU queue per slab class to track and evict the least recently used item. This works well as long as object size distribution remains static. However, this is often not true in reality. For example, if all keys start with small values that grow over time, new writes will eventually require objects to be stored in a higher slab class. However, if all memory has been allocated when this happens, there will be effectively no memory to give out. This problem is called *slab calcification* and is further explored in subsubsection 3.3.6. Memcached developed a series of heuristics to move memory between slab classes, and yet they have been shown as non-optimal [151, 231, 232, 234] and error prone [233].

To avoid slab calcification, Twemcache uses slab eviction only (Figure 3.1). This allows the evicted slab to transition into any other slab class. There are three approaches to choosing the slab to evict: choosing a slab randomly (random slab), choosing the least recently used slab (slabLRU), and choosing the least recently created slab (slabLRC). In addition to avoiding slab calcification, slab-only eviction removes two pointers from object metadata compared to Memcached. We further compare object eviction and slab eviction in section 3.5.

3.1.4 Cache Use Cases

At Twitter, it is generally recognized that there are three main use cases of Twemcache: caching for storage, caching for computation, and caching for transient data. We remark that there is no strict boundary between the three categories, and production clusters are not explicitly labeled. Thus the percentages given below are rough estimates based on our understanding of each cache cluster and its corresponding application.

Caching for Storage Using caches to facilitate reading from storage is the most common use case. Backend storage, such as databases, usually has a longer latency and a lower bandwidth than in-memory cache. Therefore, caching these objects reduce access latency, increases throughput, and shelters the backend from excessive read traffic. This use case

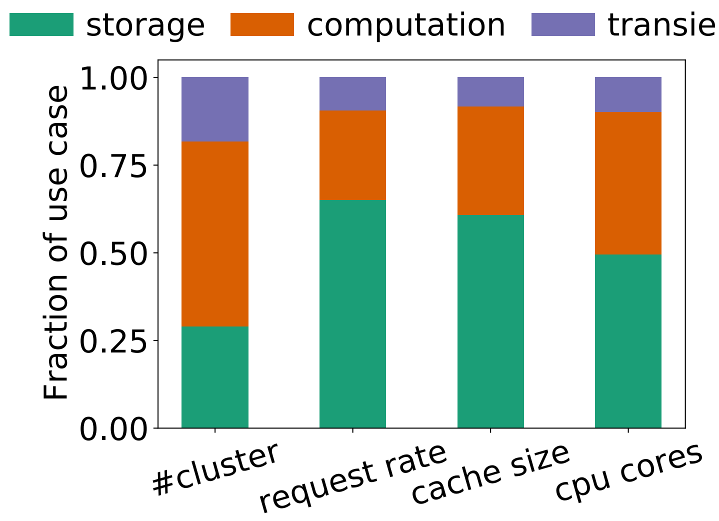


Figure 3.2: Resources consumed for the three cache use cases.

has received the most attention in research. Several efforts have been devoted to reducing miss ratio [36, 38, 48, 86, 87, 108, 154, 369] , redesigning for a denser storage device to fit larger working sets [41, 113, 320], improving load balancing [76, 80, 98] and increasing throughput [118, 203].

As shown in Figure 3.2, although only 30% of the clusters fall into this category, they account for 65% of the requests served by Twemcache, 60% of the total DRAM used, and 50% of all CPU cores provisioned.

Caching for Computation Caching for computation is not new — using DRAM to cache query results has been studied and used since more than two decades ago [10, 215]. As real-time stream processing and machine learning (ML) become increasingly popular, an increasing number of cache clusters are devoted to caching computation-related data, such as features, intermediate and final results of ML prediction, and so-called object hydration — populating objects with additional data, which often combines storage access and computation.

Overall, caching for computation accounts for 50% of all Twemcache clusters in cluster count, 26%, 31%, and 40% of request rate, cache sizes, and CPU cores.

Transient data with no backing store The third typical cache usage evolves around objects that only live in the cache, often for short periods of time. It is not caching in the strict sense, and therefore has received little attention. Nonetheless, in-memory caching is often the only production solution that meets both the performance and scalability requirements of such use cases. While data loss is still undesirable, these use cases really prize speed and tolerate occasional data loss well enough to work without a fallback.

Some notable examples are rate limiters, deduplication caches, and negative result caches. Rate limiters are counters associated with user activities. They track and cap user requests in a given time window and prevent denial-of-service attacks. Deduplication caches are a special case of rate limiters, where the cap is 1. Negative result caches store keys from a larger database that are known to be misses against a smaller, sparsely populated database. These caches short-circuit most queries with negative results and drastically reduce the traffic targeting the smaller database.

In our measurements, 20% of Twemcache clusters are under this category. Their request rates and cache sizes account for 9% and 8% of all Twemcache request rates and cache sizes. Meanwhile, they account for 10% of all CPU cores of Twemcache clusters.

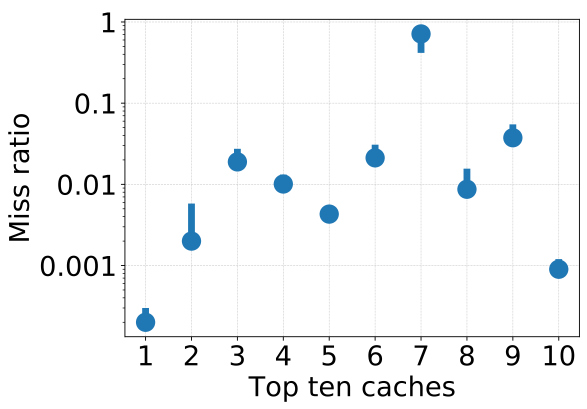
3.2 Methodology

3.2.1 Log Collection

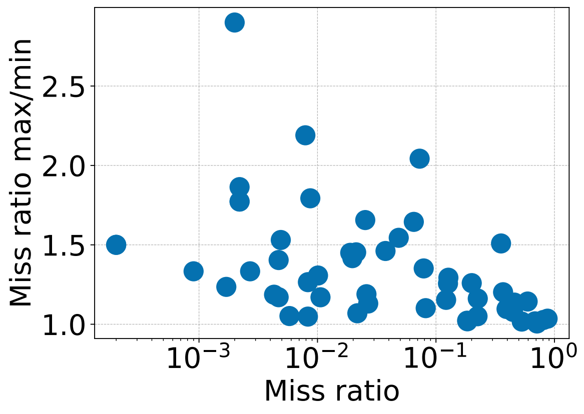
Twemcache has a built-in non-blocking request logging utility called `klog` that can keep up with designed throughput in production. While it logs one out of every 100 requests by default, we dynamically changed the sampling ratio to 100% and collected week-long *unsampled* traces from two instances of each Twemcache clusters. Collecting unsampled traces allows us to avoid drawing potentially biased conclusions caused by sampling. Moreover, we chose to collect traces from two instances instead of one to prevent possible cache failure during log collection and to compare results between instances for higher fidelity. Barring cache failures, the two instances have no overlapping keys.

3.2.2 Log Overview

We collected around 700 billion requests (80 TB in raw file size) from 306 instances of 153 Twemcache clusters, which include all clusters with a per-instance request rate of more than 1000 queries-per-sec (QPS) at the time of collection. To simplify our analysis and presentation, we focused on the 54 largest caches, which account for 90% of aggregated



(a) Production miss ratio



(b) Miss ratio variation

Figure 3.3: a) Production miss ratio of the top ten Twemcache clusters ranked by request rates, the bar shows the max and min miss ratio across one week. Note that the Y-axis is in the log scale. b) The ratio between the max and min miss ratio is small for most caches.

QPS and 76% of allocated memory. In the following sections, we use Twemcache workloads to refer to the workloads from these 54 Twemcache clusters. Although we only present the results of these 54 caches, we did perform the same analysis on the smaller caches, and they don't change our conclusions.

3.3 Production Stats and Workload Analysis

In this section, we start by describing some common production metrics to provide a foundation for our discussion, and then move on to workload analyses that can only be performed with detailed traces.

3.3.1 Miss Ratio

Miss ratio is one of the key metrics that indicate the effectiveness of a cache. Production in-memory caches usually operate at a low miss ratio with small miss ratio variation.

We present the miss ratios of the top ten Twemcache clusters ranked by request rates in Figure 3.3a where the dot shows the mean miss ratio over a week, and the error bars show the minimum and maximum miss ratio. Eight out of the ten Twemcache clusters have a miss ratio lower than 5%, and six of them have a miss ratio close to or lower than

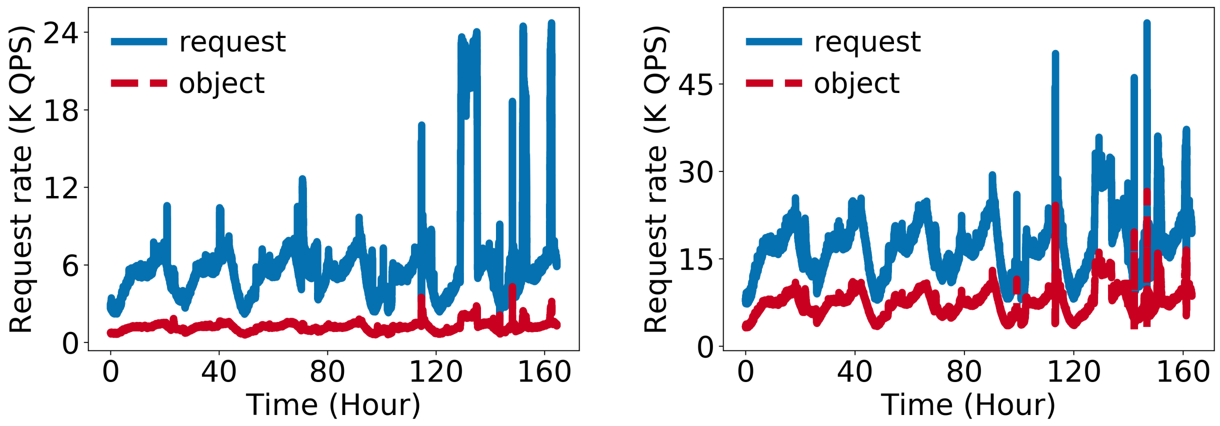


Figure 3.4: The number of requests and objects being accessed every second for two cache nodes.

1%. The only exception is a write-heavy cache cluster, which has a miss ratio of around 70% (see subsubsection 3.3.3 for details about write-heavy workloads). Compared to CDN caching [154], in-memory caching usually has a lower miss ratio.

Besides a low miss ratio, miss ratio stability is also very important. In production, it is the highest miss ratio (and request rate) that decides the QPS requirement of the backend. Therefore, a cache with a low miss ratio most of the time, but sometimes a high miss ratio is less useful than a cache with a slightly higher but stable miss ratio. Figure 3.3b shows the ratios of $\frac{mr_{max}}{mr_{min}}$ over the course of a week for different caches, where mr stands for miss ratio. We observe that most caches have this ratio lower than 1.5. In addition, the caches that have larger ratios usually have a very low miss ratio.

Low miss ratios and high stability, in general, illustrate the effectiveness of production caches. However, extremely low miss ratios tend to be less robust, which means the corresponding backends have to be provisioned with more margins. Moreover, cache maintenance and failures become a major source of disruption for caches with extremely low miss ratios. The combination of these factors indicates there's typically a limit to how much cache can reduce read traffic or how little traffic backends need to provision for.

3.3.2 Request Rate and Hot Keys

Similar to previously observed [25], request rates show diurnal patterns (Figure 3.4). Besides, spikes in request rates are also very common because the cache is the first responder to any change from the frontend services and end users.

When a request rate spike happens, a common belief is that hot keys cause the spikes [76, 155]. Indeed, load spikes often are the results of hot keys. However, we notice it is not always true. As shown in Figure 3.4, at times, when the request rate (top blue curve) spikes, the number of objects accessed in the same time interval (bottom red curve) also has a spike, indicating that the spikes are triggered by factors other than hot keys. Such factors include client retry requests, external traffic surges, scan-like accesses, and periodic tasks.

In addition to request rate spikes, caches often show other irregularities. For example, in subsubsection 3.3.6, we show that it is common to see sudden changes in object size distribution. These irregularities can happen for various reasons. For instance, users change their behavior due to a social event, the frontend service adds a new feature (or bug), or an internal load test is started.

As a critical component in the infrastructure, caches stop most of the requests from hitting the backend, and they should be designed to tolerate these workload changes to absorb the impact.

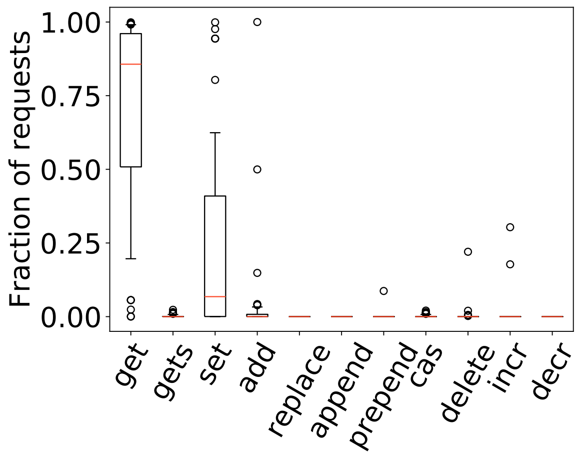
3.3.3 Types of Operations

Twemcache supports eleven different operations, of which `get` and `set` are the most heavily used by far. In addition, write-heavy cache workloads are very common at Twitter.

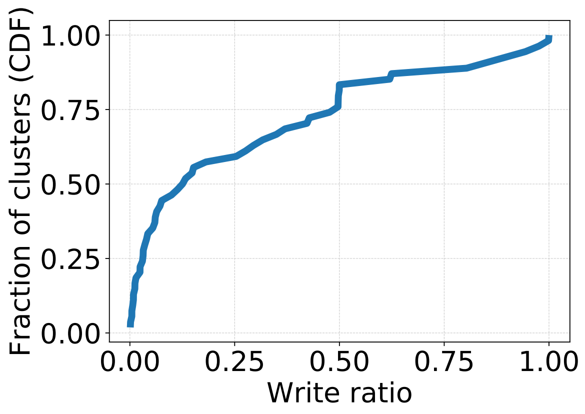
Relative usage comparison

We begin with the operations used by Twemcache workloads. Twemcache supports eleven operations `get`, `gets`, `set`, `add`, `cas` (check-and-set), `replace`, `append`, `prepend`, `delete`, `incr` and `decr`[2]. As shown in Figure 3.5a, `get` and `set` are the two most common operations, and the average `get` ratio is close to 90% indicating most of the caches are serving read-heavy workloads. Apart from `get` and `set`, operations `gets`, `add`, `cas`, `delete`, `incr` are also frequently used in Twemcache clusters. However, compared to `get` and `set`, these operations usually account for a smaller percentage of all requests. Nonetheless, these operations serve important roles in in-memory caching.

[2]See https://github.com/memcached/memcached/wiki/Commands for details about each command.



(a) Relative use of each operation



(b) Write ratio

Figure 3.5: a) The ratio of operation in each Twemcache cluster, box shows the 25^{th} and 75^{th} percentile, the red bar inside the box shows the median ratio, and whiskers are 10^{th} and 90^{th} percentile. b) write ratio distribution CDF across Twemcache clusters.

Therefore, as suggested by the author of Memcached, they should not be ignored [241].

Write ratio

Although most caches are read dominant, Figure 3.5a shows that both `get` and `set` ratios have a large range across caches. We define a workload as write-heavy if the percentage sum of `set`, `add`, `cas`, `replace`, `append`, `prepend`, `incr` and `decr` operations exceeds 30%. Figure 3.5b shows the distribution of the write ratio across caches. More than 35% of all Twemcache clusters are write-heavy, and more than 20% have a write ratio higher than 50%. In other words, in addition to the well-known use case of serving read-heavy workloads, a substantial number of Twemcache clusters are used to serve write-heavy workloads. We identify the main use cases of write-heavy caches below.

Frequently updated data. Caches under this category mostly belong to the cache for computation or transient data (subsection 3.1.4 & 3.1.4). Updates are accumulated in the cache before they get persisted, or the keys eventually expire.

Opportunistic pre-computation. Some services continuously generate data for potential consumption by itself or other services. One example is the caches storing recent user activities, and the cached data are read when a query asks for recent events from a particular user. Many services choose not to fetch relevant data on demand, but instead

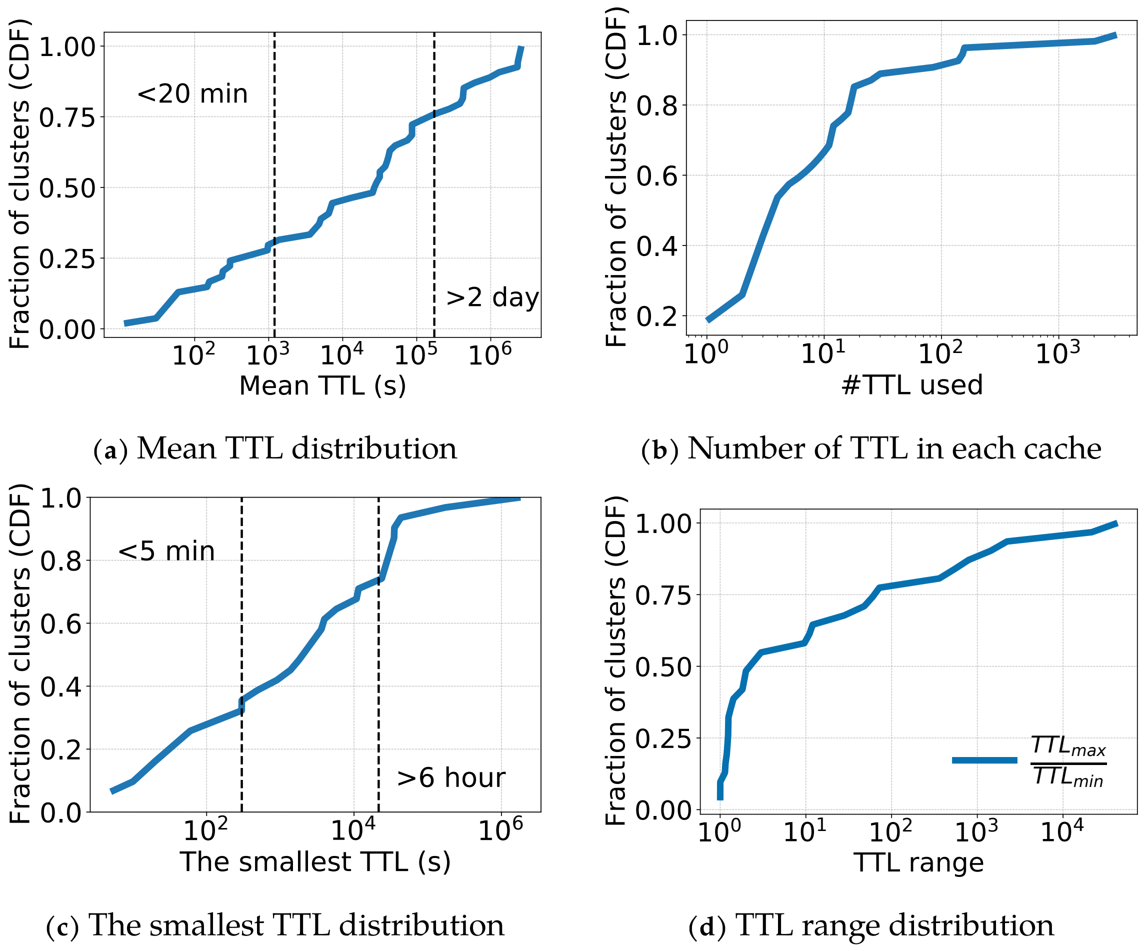
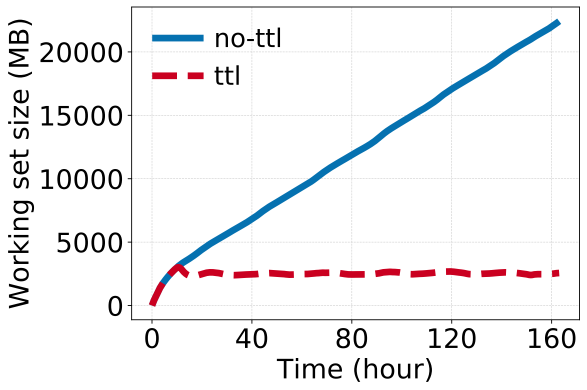
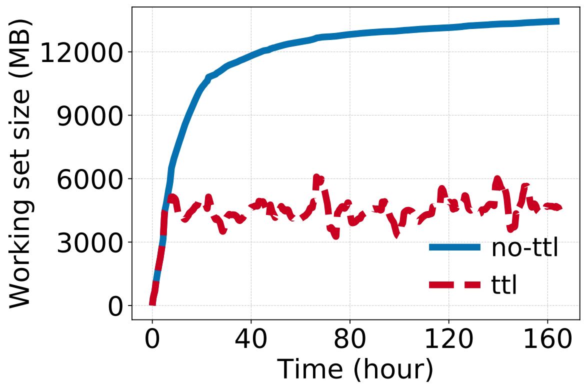


Figure 3.6: a) More than half of caches have mean TTL shorter than one day. b) Only 20% of caches use single TTL. c) The smallest TTL in each cache can be very long. d) TTL ranges in workloads are often large.

opportunistically pre-compute them for a much larger set of users. This is feasible because pre-computation often has a bounded cost, and in exchange read queries can be quickly fulfilled by pre-computed results partially or completely. Since this is a tradeoff mainly for user experience, the caches under this category see objects with fewer reuse. Therefore, the write ratio is often higher (>80%), and object access (read+write) frequency is often lower. In one case, we saw one cluster with a mean object frequency close to 1.



(a)



(b)

Figure 3.7: The working set size grows over time when TTL is not considered. However, when TTL is considered, the working set size is capped.

3.3.4 TTL

Two important features that distinguish in-memory caching from a persistent key-value store are TTL and cache eviction. While evictions have been widely studied [38, 48], TTL is often overlooked. Nonetheless, TTL has been routinely used in production. Moreover, as a response to GDPR [128], the usage of caching TTL has become mandatory at Twitter to enforce data retention policies. TTL is set when an object is first created in Twemcache and decides its expiration time. Request attempts to access an expired object will be treated as misses, so keeping expired objects in the cache is not useful.

We observe that in-memory caching workloads often use short TTLs. This usage comes from the dynamic nature of cached objects and the usage for implicit deletion. Under this condition, effectively and efficiently removing expired objects from the cache becomes necessary and important, which provides an alternative to eviction in achieving low miss ratios.

TTL Usages

We measure the mean TTLs used in each Twemcache cluster and show the TTL distribution in Figure 3.6a. The figure shows that TTL ranges from minutes to days. More than 25% of the workloads use a mean TTL shorter than twenty minutes, and less than 25% of the workloads have a mean TTL longer than two days. Such a TTL range is longer than DNS caching (minutes) [173], but shorter than common CDN object caching (days to weeks). If we divide caches into *short-TTL caches* (TTL $\leq$ 12 hours) and *long-TTL caches* (TTL $>$ 12 hours). Figure 3.6a shows 66% of all Twemcache clusters have a short mean TTL.

In addition to the mean TTL distribution, we have also measured the number of TTLs used in each cache. Figure 3.6b shows that only 20% of the Twemcache workloads use a single TTL, while the rest majority use more than one TTL. In addition, we observe that over 30% of the workloads use more than ten TTLs and there are a few workloads using more than 1000 TTLs. In the last case, some clients intentionally scatter TTLs over a pre-defined time range to avoid objects expiring at the same time. This technique is called *TTL jitter*. In another case, the clients seek the opposite effect — computing TTLs so that a group of objects will expire at the same, predetermined time.

Besides the number of TTLs used, the smallest TTL and the TTL range, defined as the ratio between TTL_{max} and TTL_{min}, are also important for designing algorithms that remove expired objects (see section 3.6). Figure 3.6c shows that the smallest TTL in each cache varies from 10s of seconds to more than half day. In detail, around 30 to 35% of the caches have their smallest TTL shorter than 300 seconds, and over 25% of caches have the smallest TTL longer than 6 hours. Figure 3.6d shows the CDF of each workload's TTL range. We observe that fewer than 40% of the workloads have a relatively small TTL range ($< 2\times$ difference), while almost 25% of the caches have $\frac{TTL_{max}}{TTL_{min}}$ over 100.

Below we present the three main purposes of TTL to better explain how TTL settings relate to the usages of the caches.

Bounding inconsistency. Objects stored in Twemcache can be highly dynamic. Because cache updates are best-effort, and failed cache writes are not always retried, it is possible that objects stored in the in-memory cache are stale. Therefore, applications often use TTL to bound inconsistency, which is also suggested in the AWS Redis documentation [282]. TTLs for this purpose usually have relatively large values, in the range of days. Some Twitter services further developed *soft TTL* to achieve a better tradeoff between data consistency and availability. The main idea of soft TTL is to store an additional, often shorter TTL as part of the object value. When the application decodes the value of a cached object and notices that the soft TTL has expired, it will refresh the cached value from its corresponding source of truth in the background. Meanwhile, the application continues to use the older value to fulfill current requests without waiting. Soft TTL is typically designed to increase with each background refresh, based on the assumption that newly created objects are more likely to see a high volume of updates and therefore inconsistency.

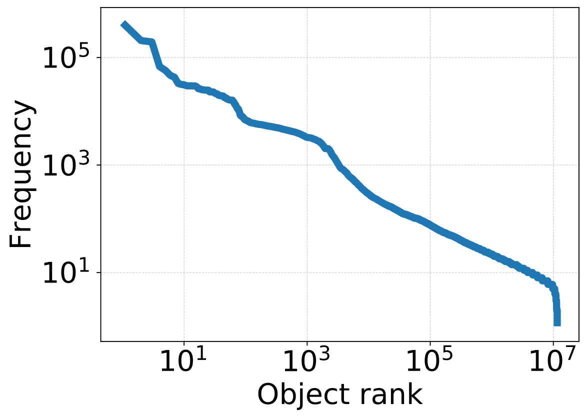
Implicit deletion. In some caches, TTL reflects the intrinsic life span of stored objects. One example is the counters used for API rate limiting, which are declared as the maximum number of requests allowed in a time window. These counters are typically stored in cache only, and their TTLs match the time windows declared in the API specification. In addition to rate limiters, GDPR-required TTL would also fall into this category, so no data would live in cache beyond the duration permitted under the law.

Periodic refresh. TTL is also used to promote data freshness. For example, a service that calculates how much a user's interest matches a cluster/community using ML models can make "who-to-follow" types of recommendations with the results. The results are cached for a while because user characteristics tend to be stable in the very short term, and the calculation is relatively expensive. Nonetheless, as users engage with the site, their portraits can change over time. Therefore such a service tends to recompute the results for each user periodically, using or adding the latest data since the last update. In this case, TTL is used to pace a relatively expensive operation that should only be performed infrequently. The exact value of the TTL is the result of a balance between computational resources and data freshness, and can often be dynamically updated based on circumstances.

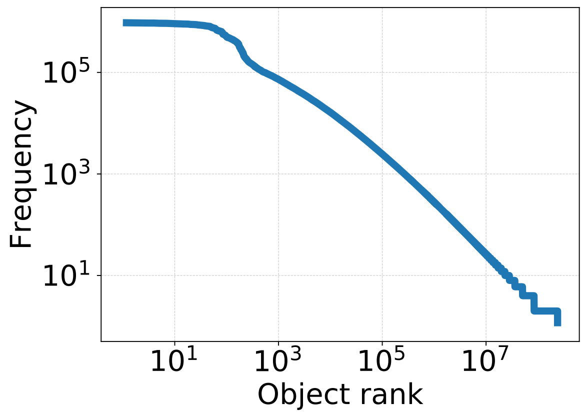
Working Set Size and TTL

Having the majority of caches use short TTLs indicate that the *effective working set size* (WSS_E) — the size of all unexpired objects should be loosely bounded. In contrast, the *total working set size* (WSS_T), the size of all active objects regardless of TTL, can be unbounded.

In our measurements, we identify two types of workloads shown in Figure 3.7. The first type (Figure 3.7a) has a continuously growing WSS_T, and it is usually related to user-generated content. With new content being generated every second, the total working set size keeps growing. The second type of workload has a large growth rate in WSS_T at first, and then the growth rate decreases after this initial fast-growing period, as shown in Figure 3.7b. This type of workload can be users related, the first quick increase corresponds to the most active users, and the slow down corresponds to less active users. Although the two workloads show different growth patterns in total working set size, the effective working set size of both arrive at a plateau after reaching its TTL. Although



(a)



(b)

Figure 3.8: Some workloads showing small deviations from Zipfian popularity. a) The least popular objects are less popular than expected. b) The most popular objects are less popular than expected.

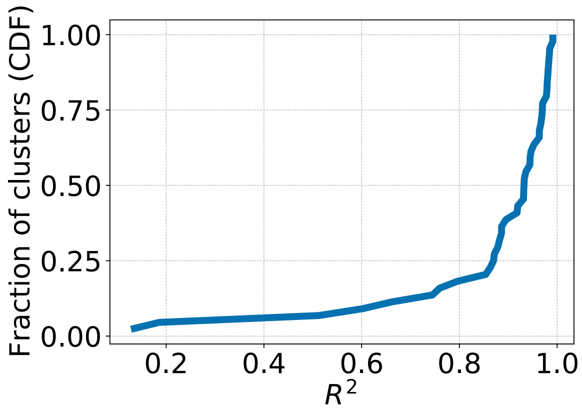
the WSS_E may fluctuate and grow in the long term, the growth rate is much slower compared to WSS_T.

Bounded WSS_E means that, for many caches, there exists a cache size that the cache can achieve the compulsory miss ratio. If an in-memory caching system can remove expired objects in time. This suggests the importance of quickly removing expired objects from the cache, especially for workloads using short TTLs. Unfortunately, while eviction has been widely studied [38, 48, 193], expiration has received little attention. And we will show in subsection 3.6.2, existing solutions fall short on expiration.

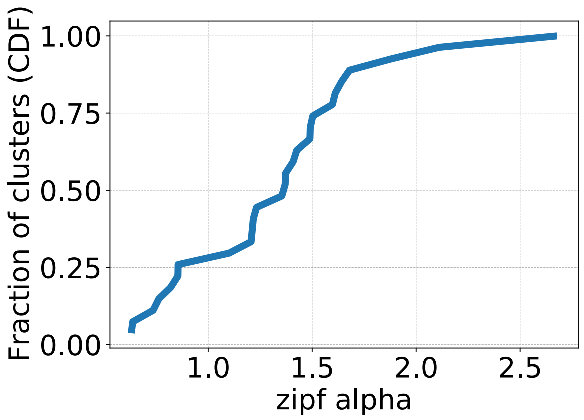
3.3.5 Popularity Distribution

Object popularity is another important characteristic of a caching workload. Popularity distribution is often used to describe the cachebility of a workload. A popular assumption is that cache workloads follow Zipfian distribution [53], and the frequency-rank curve plotted in the log-log scale is linear. A large body of work optimizes system performance under this assumption [76, 98, 117, 170, 211, 281].

Measuring all Twemcache workloads, we observe the majority of the cache workloads still follow Zipfian distribution. However, some workloads show deviations in two ways. First, unpopular objects appear significantly less than expected (Figure 3.8a), or the



(a) R^2 in Zipfian fitting



(b) Zipfian parameter α

Figure 3.9: a) Most of the workloads follow Zipfian popularity distribution with large confidence R^2. b) The parameter α in the Zipfian distribution is large, and the popularity of most workloads is highly skewed ($\alpha > 1$).

most popular objects are less popular than expected (Figure 3.8b). The first deviation happens when objects are always accessed multiple times so that there are few objects with frequency smaller than some value. The second deviation happens when the client has an aggressive client-side caching strategy so that the most popular objects are often cached by the client. In this case, the cache is no longer single-layer.

Although these deviations happen, they are rare, and we believe it is still reasonable to assume that in-memory caching workloads follow the Zipfian distribution. Since most parts of the frequency-rank curves are linear in the log-log scale, we use linear fitting[3] confidence R^2 [1] as the metric for measuring the goodness of fit. Figure 3.9a shows the results of the fitting. 80% of all workloads have R^2 larger than 0.8, and more than 50% of workloads have R^2 larger than 0.9. These results indicate that the popularity of most in-memory caching workloads at Twitter follows the Zipfian distribution. We further measure the parameter α of the Zipfian distribution shown in Figure 3.9b. The figure shows that most of the α values are in the range from 1 to 2.5, indicating the workloads are highly skewed.

[3]We remark that linear regression is not the correct way to model Zipf distribution from the view of statistics, we perform this to align with existing works [53].

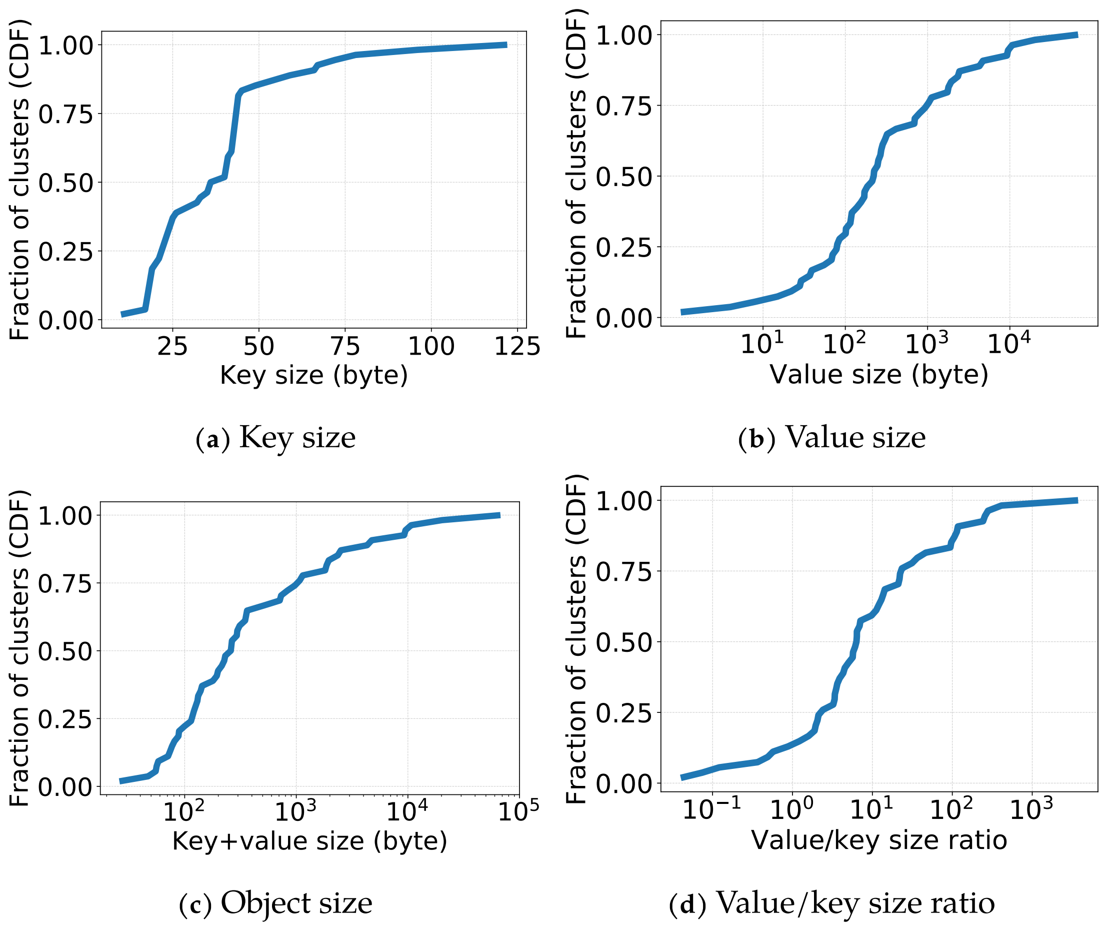


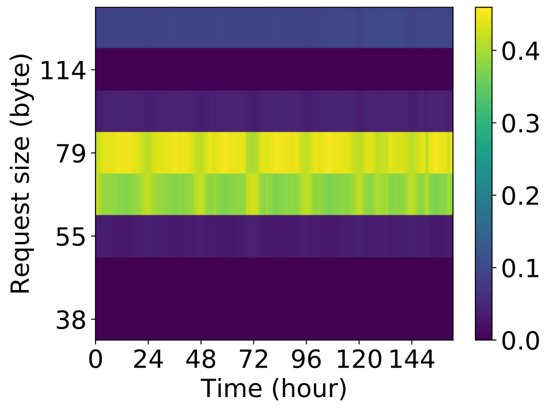
Figure 3.10: Mean key, value, object size distribution and mean $\frac{value}{key}$ size ratio across all caches.

3.3.6 Object Size

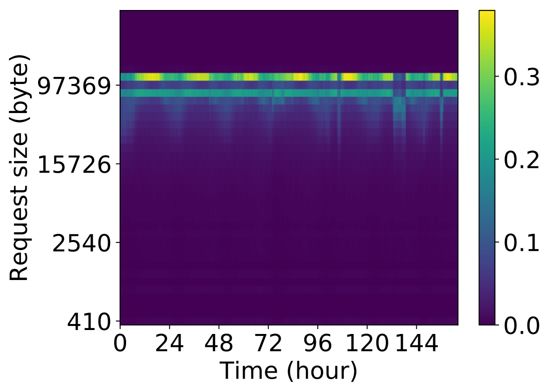
One feature that distinguishes in-memory caching from other types of caching is the object size distribution. We observe that similar to previous observations [25], the majority of objects stored in Twemcache are small. In addition, size distribution is not static over time, and both periodic distribution shifts and sudden changes are observed in multiple workloads.

Size Distribution

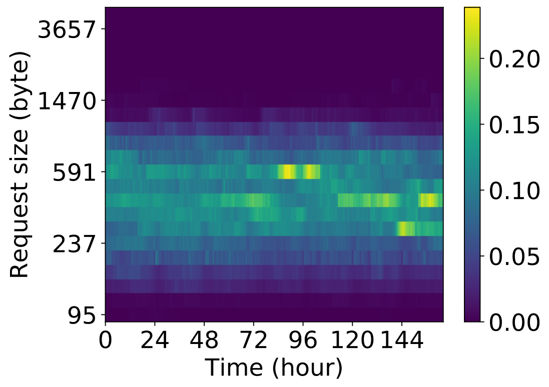
We measure the mean key size and value size in each Twemcache cluster and present the CDF of the distributions in Figure 3.10. Figure 3.10a shows that around 85% of Twemcache clusters have a mean key size smaller than 50 bytes, with a median smaller



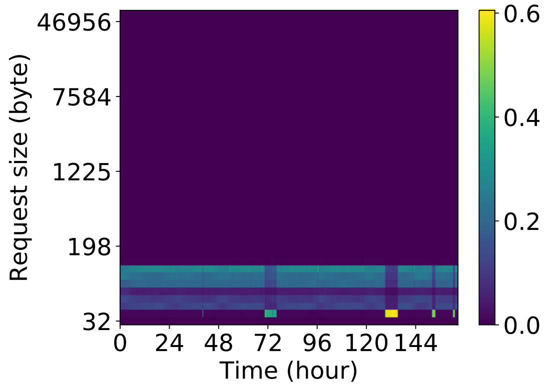
(**a**)



(**b**)



(**c**)



(**d**)

Figure 3.11: Heatmap showing request size distribution over time for four typical caches. X-axis is time, Y-axis is the object size using slab class size as bins, and the color shows the fraction of requests that fall into a slab class in that time window.

than 38 bytes. Figure 3.10b shows that the mean value size falls in the range from 10 bytes to 10 KB, and 25% of workloads show a mean value size smaller than 100 bytes, and the median is around 230 bytes. Figure 3.10c shows the CDF distribution of the mean object size (key+value), which is very close to the value size distribution except at small sizes. Value size distribution starts at size 1, while object size distribution starts at size 16. This indicates that for some of the caches, the value size is dramatically smaller than the key size. Figure 3.10d shows the ratio of mean value and key sizes. We observe that 15% of workloads have a mean value size smaller than or equal to the mean key size, and 50% of workloads have a value size smaller than 5× key size.

Size Distribution Over Time

In the previous section, we investigated the static size distribution of all objects accessed in the one-week time of each Twemcache cluster. However, the object size distribution of workloads is usually not static over time. In Figure 3.11, we show how the size distribution changes over time. The X-axis shows the time, and the Y-axis shows the size of objects (using slab class size as bins), the color shows how many of the objects in one time window fall into each slab class. We observe that some of the workloads show diurnal patterns (Figure 3.11a, 3.11b), while others show changes without strict patterns.

Periodic/diurnal object size shifts can come from the following sources, a) value for the same key grows over time. and b) size distribution correlates with temporal aspects of key access. For example, text content generated by users in Japan is shorter/smaller than those by users in Germany. In this case, it is the geographical locality that drives the temporal pattern. On the other hand, we do not yet have a good understanding of how most sudden, non-recurring changes happen. Current guesses include user behavior changes during events and a temporary change in production settings.

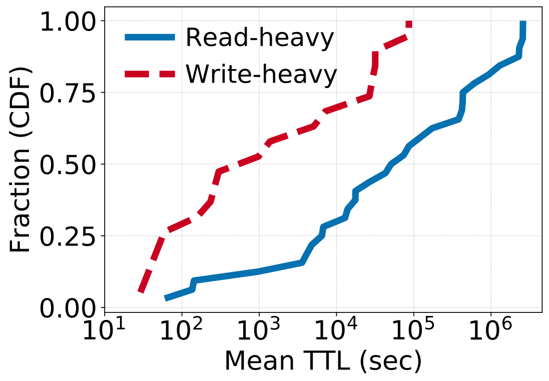
Both short-term and long-term size distribution shifts pose additional challenges to memory management in caching systems. They make it hard to control or predict external fragmentation in caches that use heap memory allocators directly, such as Redis. For slab-based caching systems, such changes can cause slab calcification. In subsection 3.6.5, we discuss why existing techniques do not completely address the problem.

3.4 Further Analysis of Workload Properties

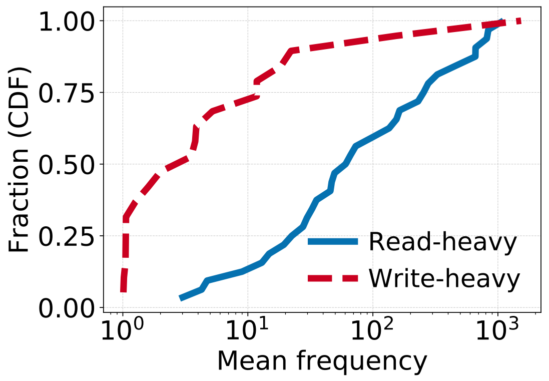
We have shown the properties of the in-memory caching workloads at Twitter. In this section, we show the relationship between the properties, and how they relate to major caching use cases.

3.4.1 Correlations between Properties

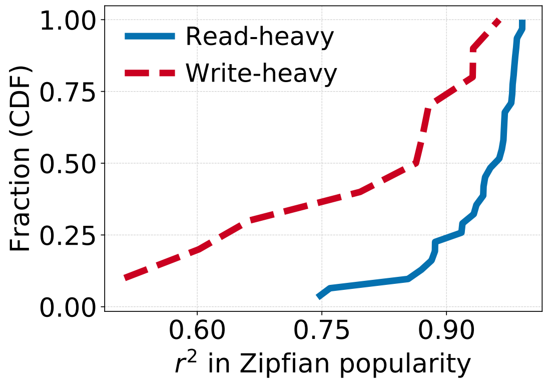
Throughout the analysis in previous sections, we observe some workload characteristics have strong correlations with the write ratio. For example, write-heavy workloads usually use short TTLs. Presented in Figure 3.12a, the dashed red curve shows the mean TTL distribution of write-heavy workloads, and the solid blue curve shows the mean TTL



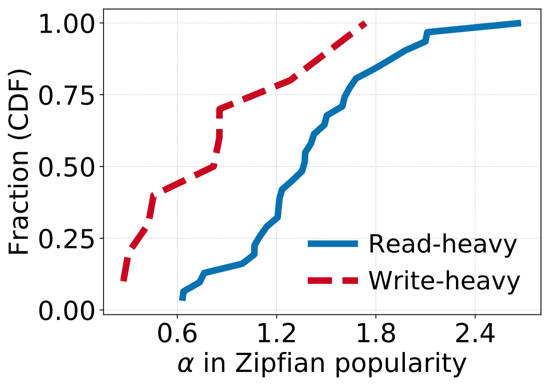
(**a**) TTL distribution



(**b**) Object frequency distribution



(**c**) R^2 in fitting Zipfian



(**d**) α of Zipfian distribution

Figure 3.12: Write-heavy workloads tend to show short TTL, small object access frequency, relatively large deviations from Zipfian popularity distribution, and are usually less skewed (small α).

distribution of read-heavy workloads. Around 50% of the write-heavy workloads have mean TTL shorter than 10 minutes, while for read-heavy workloads, this is 15 hours. Further, the Pearson coefficient between write ratio and log [4] of mean TTL (Table 3.1) is -0.63 indicating a negative correlation, confirming that large write ratio workloads usually have short TTLs.

Besides TTL, write-heavy workloads also show low object frequencies. We present the mean object frequency (in terms of the number of accesses in the traces) of read-heavy and write-heavy workloads in Figure 3.12b. It shows that read-heavy workloads have a mean frequency mostly in the range from 6 to 1000, with 75% percentile above 200. Meanwhile, write-heavy workloads have a mean frequency mostly between 1 and 100, with 75%

[4]We choose to use log of TTL and frequency because of their wide ranges in different workloads.

Table 3.1: Correlation between write ratio and other properties

Property	Pearson coefficient with write ratio
$\log$(TTL)	-0.6336
$\log$(Frequency)	-0.7414
Zipf fitting R^2	-0.7690
Zipf alpha	-0.7329

percentile below 10. We further confirm this relationship with the Pearson coefficient between write ratio and $\log$ of frequency, which is -0.7414 (Table 3.1), suggesting the low object access frequency in write-heavy caches.

In addition, the popularity of write-heavy workloads has relatively larger deviations from Zipfian distribution, and the fitting confidence R^2 is usually much smaller than that of read-heavy workloads (Figure 3.12c). Moreover, the α parameter of Zipfian distribution in write-heavy workloads is usually small, as shown in Figure 3.12d. It shows the write-heavy workloads have a median α around 0.9, and the median of read-heavy workloads have an α around 1.4.

3.4.2 Properties of Different Cache Use Cases

Here we further explore common properties exhibited by each of the three major caching use cases as described in subsection 3.1.4.

Caching for Storage

Caches for storage usually serve ready-heavy workloads, and their popularity distributions typically follow the Zipfian distribution with a large parameter α in the range of 1.2 to 2.2. While this type of workload is highly skewed, they are easier to cache, and in production, 95% of these clusters have miss ratios of around or less than 1%. Being more cacheable and having smaller miss ratios do not indicate they have small working set sizes. In our observation, 7 of the top 10 caches (ranked by cache size) belong to this category.

Because these caches store objects persisted in the backend storage, any modifications to the objects are explicitly written to both the backend and the cache. Therefore the TTLs used in these caches are usually large, in the range of days. There is no specific pattern for object size in this type of cache, and the value can be as large as tens of KB, or as small as a few bytes. For example, the number of favorites a tweet received is persisted in the backend database and sometimes cached.

Caching for Computation

Caches under this category serve both read-heavy and write-heavy traffic depending on the workloads. For example, machine learning feature workloads are usually read-heavy showing a good fit of Zipfian popularity distribution. While intermediate computation workloads are normally write-heavy and show deviations from Zipfian. Compared to caching for storage, workloads under this category use shorter TTLs, usually determined by the application requirement. For example, caches storing intermediate computation data usually have TTLs of no more than minutes because other services will consume the data in a short time. For features and prediction results, the TTLs are usually in the range of minutes to hours (some up to days) depending on how fast the underlying data change and how expensive the computation is. The mean TTLs we observe for caches under this category is 9.6 hours. There are no particular patterns in object sizes in these caches.

Since objects stored in these caches are indirectly related to users and contents, the workloads usually have large key spaces and total working set sizes. For example, a cache storing the distance between two users will require a N^2 cache size where N denotes the number of users. However, because these caches have short TTLs, the effective working set sizes are usually much smaller. Thus removing expired objects can be more important than eviction for these caches.

As real-time stream processing becomes more popular, we envision there will be more caches being provisioned for caching computation results. Because the characteristics are different from caching for storage, they may not benefit equally from optimizations that only aim to make the read path fast and scalable, such as optimistic cuckoo hashing [118]. Therefore, including evaluation against caching-for-computation workloads that are write-heavy and more ephemeral will paint a more complete picture of the capabilities of any caching system.

Transient Data with No Backing Store

There are two characteristics associated with this type of cache: Caches under this category usually have short TTLs, and the TTLs are often used to enforce implicit object deletion (subsection 3.3.4). In addition, objects in these caches are usually tiny and we observe an average object size of 54 bytes. Although caches of this type only contribute 9% of total Twemcache clusters request rate and 8% of total cache sizes, they currently play an irreplaceable role in site operations.

3.5 Eviction Algorithms

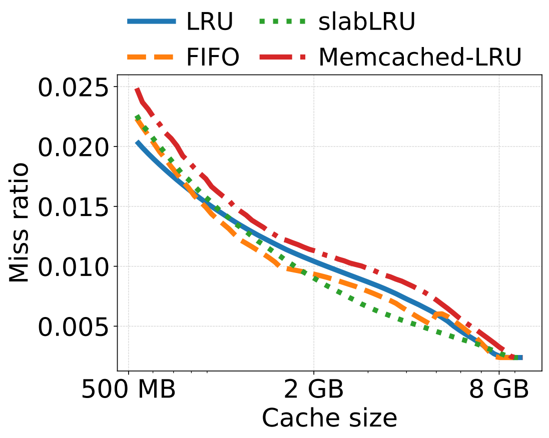
We have shown the characteristics of in-memory cache workloads in the previous sections. In this section, we use the same cache traces to investigate the impact of eviction algorithms. This evaluation considers production algorithms offered by Twemcache and other production systems.

3.5.1 Eviction algorithm candidates

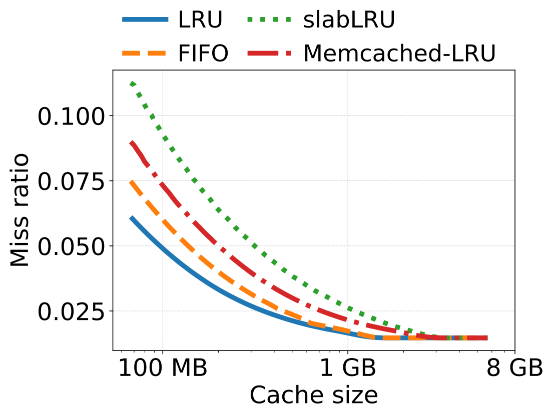
Object LRU and object FIFO. LRU and FIFO are the most common algorithms used in production caching systems [14, 328]. However, they cannot be applied to systems using slab-based memory management, such as Twemcache, without modification. Therefore, we evaluate LRU and FIFO assuming the workloads are served using a non-slab-based caching system while ignoring memory inefficiency caused by external fragmentation. As a result, we expect the results to have a bias toward the effectiveness of LRU and FIFO compared to the three slab-based algorithms. Production results for these two algorithms might be worse than what is suggested in this section, depending on the workloads.

slabLRU and slabLRC. These two algorithms are part of eviction algorithms offered in Twemcache. slabLRU and slabLRC are equivalent to LRU and FIFO but executed at a level much coarser granularity of slabs rather than a single object. Twitter employs these algorithms to alleviate the effect of slab calcification and also to reduce the size of per-object metadata.

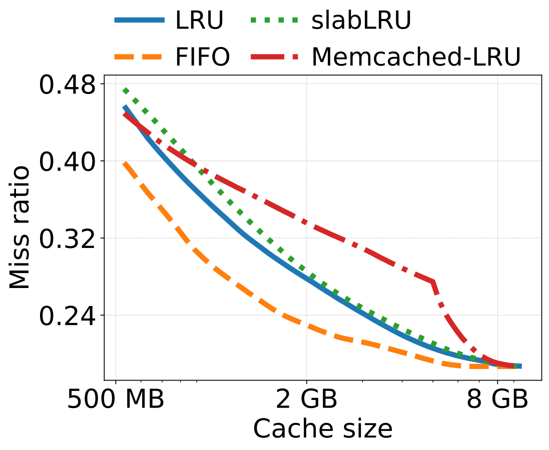
Random slab eviction. Besides slabLRU and slabLRC, Twemcache also offers Random slab eviction, which globally picks a random slab to evict. This algorithm is workload-agnostic with robust behavior, and therefore used as the default policy in production.



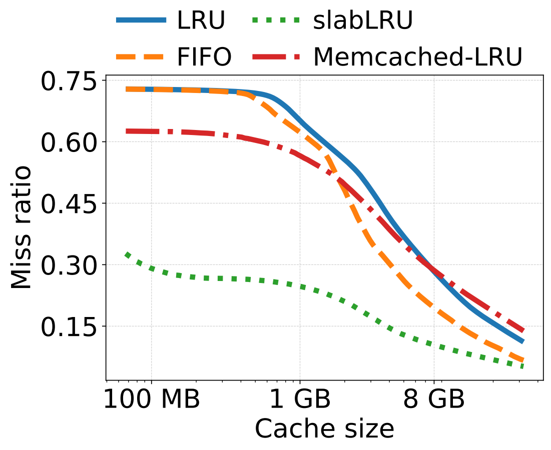
(a) Similar miss ratio



(b) LRU is better



(c) FIFO is better

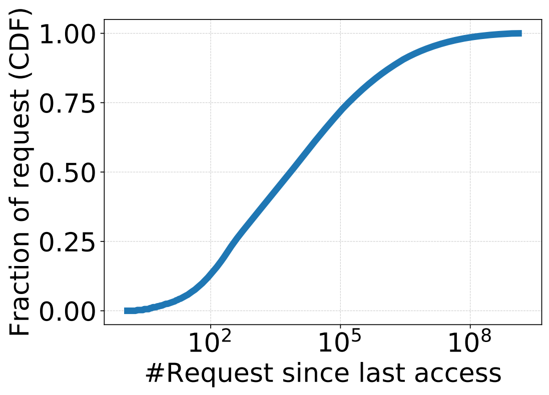


(d) slabLRU is better

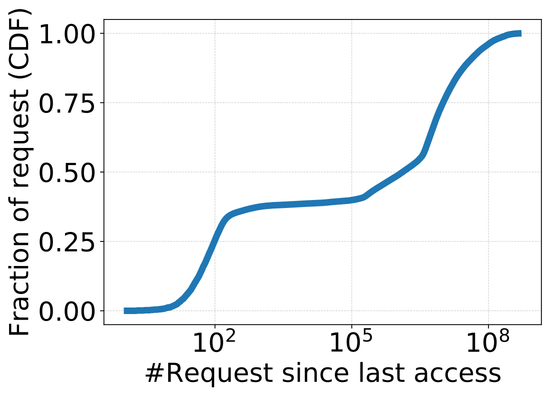
Figure 3.13: Four typical miss ratio results: a) all algorithms have similar performance, b) LRU is slightly better than others, c) FIFO is better than others, d) slabLRU is much better than others.

However, Random is rarely the best of all algorithms and is non-deterministic. Therefore, we do not include it in comparison.

Memcached-LRU. Memcached adapted LRU by creating one LRU queue per slab class. We call the resulting eviction algorithm Memcached-LRU, which does not enable Memcached's slab auto-move functionality. We did, however, evaluate Memcached-LRU with slab auto-move turned on, and most of the results are somewhere between LRU and slabLRU. The rest of the paper omits this combination.



(a)



(b)

Figure 3.14: The inter-arrival gap distribution corresponding to the workloads in Figure 3.13b and Figure 3.13c respectively.

3.5.2 Simulation Setup

We built an open-source simulator called libCacheSim [368] to study the steady-state miss ratio of the different eviction algorithms. Specifically, we use five-day traces to warm up the caches, then use one-day traces to evaluate cache miss ratios. Each algorithm is applied against all traces and then grouped by results.

In terms of cache sizes, our simulation always starts with 64MB of DRAM and chooses the maximum as 2× their current memory in production. We stop increasing the size for a particular workload when all algorithms have reached the compulsory miss ratio. Note that when plotting, the size range is truncated to better present the trend.

3.5.3 Miss Ratio Comparison

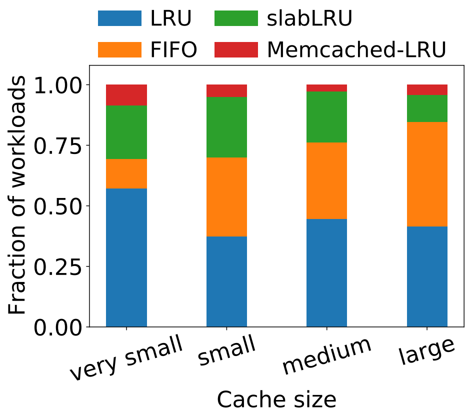
The outcome of our comparison can be grouped into four types, and representatives of each are shown in Figure 3.13.

The first group shows comparable miss ratios for all algorithms in the cache sizes we evaluated. For this type of workload, the choice of eviction algorithms has a limited impact on the miss ratio. Production deployments may very well favor simplicity or decide based on other operational considerations such as memory fragmentation. Twemcache uses random slab eviction by default because random eviction is simple and requires less metadata.

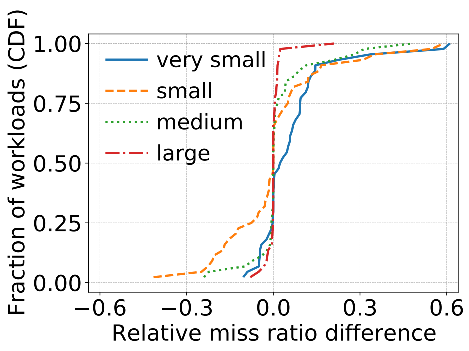
The second type of result shows that for some workloads, LRU works better than others. Such a result is often expected because LRU protects recently accessed objects and is well-known for its miss ratio performance in workloads with the strong temporal locality.

The third type of result shows that FIFO is the best eviction algorithm (Figure 3.13c). This result is somewhat surprising since it does not conform to what is typically observed in caching of other scenarios such as CDN caching. We give our suspected reasons below. Figure 3.14 shows the inter-arrival time distribution of the two workloads in Figure 3.13b and Figure 3.13c respectively. The inter-arrival time is the number of requests between two accesses to the same object. Figure 3.14a shows a smooth inter-arrival time curve, while Figure 3.14b shows a curve with multiple segments. For workloads with inter-arrival time like Figure 3.14a, LRU can work better than FIFO because it promotes recently accessed objects, which have a higher chance of being reused soon. This promotion protects the recently accessed objects but demotes other objects that are not reused recently. Demoting non-recently used objects can be an unwise decision if some of the demoted objects will be reused after 10^6 requests, such as the ones shown in Figure 3.14b. In contrast, FIFO treats each stored object equally; in other words, it protects the objects with a large inter-arrival gap. Therefore, for workloads similar to the one in Figure 3.14b, FIFO can perform better than LRU. Such workloads may include scan type of requests such as a service that periodically sends emails.

The last type of result shows that in some workloads, slabLRU performs much better than any other algorithms. The main reason is that the workloads showing this type of result have periodic/diurnal changes. Figure 3.11b shows the object size distribution over time of the workload corresponding to Figure 3.13d. We suspect this is due to the following reason, but we leave the verification as future work. Although LRU and FIFO are not affected by any change in object size distribution, they cannot respond to workload change instantly. In contrast, slabLRU can quickly adapt to a new workload when the new workload uses a different slab class because it prioritizes the slabs that have more recent access. From another view, slabLRU gives a larger usable cache size for the new workloads (slab class). Figure 3.13d shows that the difference between algorithms reduces at larger cache sizes, this is because the benefit of having a large usable cache size diminishes as cache size increases. Moreover, in these workloads, Memcached-LRU sometimes has



(**a**) The best eviction algorithms.



(**b**) The relative miss ratio difference between FIFO and LRU. Positive means FIFO is worse.

Figure 3.15: Miss ratio evaluation under different cache sizes.

better performance than LRU, but for most of the workloads, Memcached-LRU is worse (not shown in the figure) because of the missing capability of moving slabs. Thus it has a smaller usable cache size. When Memcached-LRU has better performance at small cache sizes, we suspect that the changing workloads cause thrashing for LRU and FIFO [36]. Since Memcached-LRU can only evict objects from the same slab class as the new object, it protects the objects in other slab classes from thrashing, thus showing better performance.

In most cases, both miss ratio and the difference between algorithms decrease as cache capacity increases. We observe that within our simulation configuration, which stops at or before 2× current size, the difference between algorithms eventually disappears. This suggests that to achieve a low miss ratio in real life, it can be quite effective to create implementations that increase the effective cache capacity, such as through metadata reduction, adopting higher capacity media, or data compression.

Given there are more than a couple of workloads showing each of the four result types, we would like to explore whether there is one algorithm that is often the best or close to the best most of the time.

In the next section, we explore how often each algorithm is the best with a special focus on LRU and FIFO.

3.5.4 Aggregated Statistics

In this section, we evaluate the same set of algorithms as in subsection 3.5.3, focusing on four distinct cache sizes and presenting the aggregated statistics. Because different workloads have different working set sizes and compulsory miss ratios, we choose the four cache sizes in the following way. We define the *ultimate cache size* s_u to be the size where LRU achieves compulsory miss ratio for a workload. However, if LRU can not achieve compulsory miss ratio at 2× production cache size, we use 2× production cache size as s_u. We choose *large* cache size to be 90% of s_u, and *medium*, *small* and *very small* cache sizes to be 60%, 20% and 5% of s_u respectively. We remark that, at Twitter, 76% of the caches have cache sizes larger than the *large* cache size category, and 34% of the rest have cache sizes within 10% of the *large* cache size.

We show the miss ratio comparison in Figure 3.15a, where each bar shows the fraction of workloads for which a particular algorithm is the best. We see that at the *large* cache size, slabLRU is the best for around 10% of workloads, and this fraction gradually increases as we reduce cache size. This increase is because for smaller cache sizes, quickly adapting to workload change is more valuable. Besides this, FIFO has similar performance compared to LRU at *small, medium, and large* size categories. And only at *very small* cache sizes, LRU becomes significantly better than FIFO. This is because at relatively large cache sizes, promoting recently accessed objects is less crucial. Instead, not demoting other objects is more helpful in improving the miss ratio, especially for workloads having multiple segments in inter-arrival time like the one shown in Figure 3.14b.

Figure 3.15a suggests of the workloads, FIFO is as good as LRU at reasonably large cache sizes. Now we explore the magnitude by which FIFO is better or worse compared to LRU on each workload. Figure 3.15b shows the relative miss ratio difference between FIFO and LRU: $\left(\frac{mr_{\text{FIFO}} - mr_{\text{LRU}}}{mr_{\text{LRU}}}\right)$, where mr stands for miss ratio, for each workload at different cache sizes. When the value on X-axis is positive, it indicates that FIFO has a higher miss ratio, and LRU has better performance, while a negative value indicates the opposite. We observe that all the curves except the one for a very small cache size are all close to being symmetric around the x-axis value 0. This indicates that across workloads, FIFO and LRU have similar performance for small, medium, and large cache sizes. For the very small size category, we observe LRU being significantly better than FIFO, this is because for workloads with temporal locality, promoting recently accessed objects becomes crucial at very small cache sizes. In production, most of the caches are running at cache sizes larger than or close to the *large* category. We believe that for most in-memory caching workloads, FIFO and LRU have similar performance at reasonably large cache sizes.

The fact that FIFO and LRU often exhibit similar performance in production-like settings is important because using LRU usually incurs extra computational and memory overhead compared to FIFO [199, 203]. For example, implementing LRU in Memcached requires extra metadata and locks, some of which can be removed if FIFO is used.

3.6 Implications

In this section, we show how our observations differ from previous work, and what the takeaways are for informing future in-memory caching research.

3.6.1 Write-heavy Caches

Although 70% of the top twenty Twemcache clusters serve read-heavy workloads (subsubsection 3.3.3), write-heavy workloads are also common for in-memory caching. This is not unique to Twitter. Previous work [25] from Facebook also pointed out the existence of write-heavy workloads, although their prevalence of them was not discussed due to the limited number of workloads. Furthermore, write-heavy workloads are expected to increase in prominence as the use case of caching for computation increases (subsection 3.1.4). However, most of the existing systems, optimizations, and research assume a read-heavy workload.

Write-heavy workloads in caching systems usually have lower throughput and higher latency, because the write path usually involves more work and can trigger more expensive events such as eviction. In Twitter's production, we observe that serving write-heavy workloads tend to have higher tail latencies. Scaling writes with many threads tends to be more challenging as well. In addition, as discussed in section 3.4, write-heavy workloads have shorter TTLs with less skewed popularity, which are in sharp contrast to read-heavy workloads. This calls for future research on designing systems and solutions that consider performance on write-heavy workloads.

3.6.2 Short TTLs

In subsubsection 3.3.4, we show that in-memory caching workloads frequently use short TTLs, and the usage of short TTLs reduces the effective working set size. Therefore, removing expired objects from the cache is far more important than evictions in some cases. In this section, we show that existing techniques for proactively removing expired objects (termed proactive expiration) are not sufficient. This calls for future work on better proactive expiration designs for in-memory caching systems.

Transient object cache. An approach employed for proactive expiration (especially for handling short TTLs), proposed in the context of in-memory caches at Facebook [259], is to use a separate memory pool (called transient object pool) to store short-lived objects. The transient object cache consists of a circular buffer of size t with the element at index i being a linked list storing objects expiring after i seconds. Every second, all objects in the first linked list expire and are removed from the cache, then all other linked lists advance by one.

This approach is effective only when the cache user uses a mix of very short and long TTLs with the short TTL usually in the range of seconds. Since objects in the transient pool are never evicted before expiration, the size of the transient pool can grow unbounded and cause objects in the normal pool to be evicted . In addition, the TTL threshold of admitting into a transient object pool is non-trivial to optimize.

As we show in Figure 3.6b, 20% of the Twemcache workloads use a single TTL. For these workloads, the transient object pool does not apply. For the workloads using multiple TTLs, we observe that fewer than 35% have their smallest TTL shorter than 300 seconds, and over 25% of caches have the smallest TTL longer than 6 hours (Figure 3.6c). This indicates that the idea of a transient object cache is not applicable to a large fraction of Twemcache clusters.

Background crawler. Another approach for proactive expiration, which is employed in Memcached, is to use a background crawler that proactively removes expired objects by scanning all stored objects.

Using a background crawler is effective when TTLs used in the cache do not have a broad range. While scanning is effective, it is not efficient. If the cache scans all the objects every T_{pass}, an object of TTL t can be scanned up to $1 + \lceil \frac{t}{T_{pass}} \rceil$ times before removal, and can overstay in the system by up to T_{pass}. The cache operator, therefore, has to make a tradeoff between wasted space and the additional CPU cycles and memory bandwidth needed for scanning. This tradeoff gets harder if a cache has a wide TTL range, which is common as observed in subsection 3.3.4. While the Twemcache workloads are single tenant, the wide TTL range issue would be further exacerbated for multi-tenant caches.

Figure 3.6d shows that TTLs used within each workload have a wide range. Close to 60% of workloads have a maximum TTL more than twice as long as the minimum and 25% of workloads show a ratio at or above 100. This indicates that for the 25% of caches if we want to ensure all objects are removed within $2\times$ their TTLs, objects with the longest TTL will be scanned 100 times before expiration.

The combination of transient object cache with background crawler could extend the coverage of workloads that can be efficiently expired. However, the tradeoff between wasted space and the additional CPU cycles and memory bandwidth consumed for scanning would still remain. Hence, future innovation is necessary to fundamentally address use cases where TTLs exhibit a broad range.

3.6.3 Highly Skewed Object Popularity

Our work shows that the object popularity of in-memory caching can be far more skewed than previously shown [41], or compared to studies on web proxy workloads [53] and CDN workloads [154]. We suspect this has a lot to do with the nature of Twitter's product, which puts great emphasis on the timeliness of its content. It remains to be seen whether this is a widespread pattern or trend. Cache workloads are also more skewed compared to NoSQL databases such as RocksDB [89], which is not surprising because database traffic is often already filtered by caches, and has the most skewed portion removed via cache hits. In other words, in-memory caching and NoSQL databases often observe different traffic even for the same application. Besides these two reasons, sampling sometimes results in bias in the popularity modeling, and we avoid this by collecting unsampled traces. Our observation that the workloads still follow Zipfian distribution with large alpha value emphasizes the importance of addressing load imbalance [117, 211, 281].

3.6.4 Object Size

Similar to previously reported [25], we observe that objects cached in in-memory caching are often tiny (subsection 3.3.6). As a result, in-memory caches are not always bound by memory size; instead, close to 20% of the Twemcache clusters are CPU-bound.

On the other hand, small objects signify the relatively large overhead of metadata. Memcached stores 56-byte with each object, and Twitter's current production cache uses 38-byte metadata with each object. Reducing object metadata further can yield substantial benefits for caching tiny objects.

In addition, we observe that compared to value size, the key size can be large in some workloads. For 60% of the workloads, the mean key size and mean value size are in the same order of magnitude. This indicates that reducing key size can be very important for these workloads. Many workloads we observed have namespaces as part of the object keys, such as `NS1:NS2:...:id`. This format is commonly used to mirror the naming in a multi-tenant database, which is also observed at Facebook [66]. Namespaces thus can occupy large fractions of precious cache space while being highly repetitive within a single cache cluster. However, there are no known techniques to "compress" the keys. To encourage and facilitate future research on this, we keep the original but anonymized namespace in our open-sourced traces.

Several recent works [38, 48] on reducing miss ratio (improving memory efficiency) focused on improving eviction algorithms and often adding more metadata. Given our observations here, we would like to call more attention to the optimization of cache metadata and object keys.

3.6.5 Dynamic Object Size Distribution

In subsubsection 3.3.6, we show that the object size distribution is not static, and the distribution shifts over time can cause out-of-memory (OOM) exceptions for caching systems using external allocators, or slab calcification for those using slab-based memory management. In order to solve this problem, one solution, employed by Facebook, is to migrate slabs between slab classes by balancing the age of the oldest items in each class [259]. Earlier versions of Memcached approached this problem by balancing the eviction rate of each slab class. Since version 1.6.6, Memcached has also moved to use the solution of balancing the age as mentioned above.

Besides efforts in production systems, slab assignment, and migration have also been a hot topic in recent research [61, 85, 86, 151]. However, to the best of our knowledge, the problem has only been studied under a "semi-static" request sequence. Specifically, the research so far assumes that the miss ratio curve or some other properties of each slab class hold steady for a certain amount of time, which often precludes periodic and sudden changes in object size distribution.

In general, the temporal properties of object sizes in the cache are not well understood or quantified. As presented in Figure 3.11c and Figure 3.11d, it is not rare to see unexpected changes in size distribution only lasting for a few hours. Sometimes it is hard to pinpoint the root cause of such changes. Nonetheless, we believe that temporal changes related to object size, whether recurring or as a one-off, usually have drivers with roots beyond the time dimension. For example, the tweet size drift throughout the day may very well depend on the locales or geo-location of active users. Some caches may be shared by datasets that differ in size distribution and access cycles, resulting in different distributions dominating the access pattern at different instants of the day. In this sense, studying the object size distribution over time could very well provide deeper insights into the characteristics of the datasets being cached. Considering the increasing interest in using machine learning and other statistical tools to study and predict caching behavior, we think object size dynamics might provide a good proxy to evaluate the relationship between basic dataset attributes and their behavior in the cache, allowing caching systems to make smarter decisions over time.

3.7 Related Work

Due to the nature of this work, we have discussed several related works in detail throughout the chapter.

Multiple caching and storage system traces were collected and analyzed in the past [25, 29, 66, 154, 155, 259]; however, only a limited number of reports focus on in-memory caching workloads [25, 155, 259]. The closest work to our analysis is Facebook's Memcached workload analysis [25], which examined five Memcached pools at Facebook. Similar to the observations in this work [25], we observe the sizes of objects stored in

Twemcache are small, and diurnal patterns are common in multiple characteristics. After analyzing 153 Twemcache clusters at Twitter, in addition to previous observations [25], we show that write-heavy workloads are popular. Moreover, we focus on several aspects of in-memory caching that have not been studied to the best of our knowledge, including TTL and cache dynamics. Although previous work [25] proposed analytical models on the key size, value size, and inter-arrival gap distribution, the models do not fully capture all the dimensions of production caching workloads such as changing working set and dynamic object size distribution. Compared to synthetic workload models, the collection of real-world traces that we collected and open-sourced provide a detailed picture of various aspects of the workloads of production in-memory caches.

Besides workload analysis on Memcached, there have been several workload analyses on web proxy [20, 21, 24, 307] and CDN caching [154, 344]. The photo caching and serving infrastructure at Facebook has been studied [154], with a focus on the effect of layering in caching along with the relationship between content popularity, age, and social-network metrics.

In addition to caching in web proxies and CDNs, the effectiveness of caching is often discussed in workload studies [29, 265] of file systems. However, these works primarily studied the cache to the extent of its effectiveness in reducing traffic to the storage system rather than on aspects that affect the design of the cache itself. Besides, file system caching is different from distributed in-memory caches due to a variety of reasons. For example, a file system cache usually stores objects of fixed-sized chunks (512 bytes, 4 KB or larger), while in-memory caches store objects of a much wider size range (subsection 3.3.6). Moreover, scans are common in file systems, while rare in in-memory key-value caches. Because of the similarities in the interface, in-memory caching is sometimes discussed together with key-value databases. Three different RocksDB workloads [66] at Facebook have been studied in depth, with a focus on the distribution of key and value sizes, locality, and diurnal patterns in different metrics. Although Twemcache and RocksDB have a similar key-value interface, they are fundamentally different because of their design and usage. RocksDB stores data for persistence, while Twemcache stores data to provide low latency and high throughput without persistence. In addition, compared to RocksDB, TTL, and evictions are unique to in-memory caching.

3.8 Chapter Summary

In-memory caching systems such as Memcached [228] and Redis [283] are heavily used by modern web applications to reduce access to storage and avoid repeated computations. Their popularity has sparked a lot of research, such as reducing miss ratio [38, 48, 86, 87], or increasing throughput and reducing latency [31, 118, 189, 203]. On the other hand, the effectiveness and performance of in-memory caching can be workload-dependent. And several important workload analyses against production systems [25, 259] have guided the explorations of performance improvements with the right context and tradeoffs in the past decade.

Nonetheless, there remains a significant gap in the understanding of current in-memory caching workloads. Firstly, there has been a lack of comprehensive studies covering the wide range of use cases in today's production systems. Secondly, there have been new trends in in-memory caching usage since the publication of previous work [25]. Thirdly, some aspects of in-memory caching received little attention in the existing studies, but are known as critical to practitioners. For example, TTL is an important aspect of configuring in-memory caching, but it has largely been overlooked in research. Last but not least, unlike other areas where open-source traces [284, 285, 349, 350] or benchmarks [89] are available, there has been a lack of open-source in-memory caching traces. Researchers have to rely on storage caching traces [38], key-value database benchmarks [118, 203] or synthetic workloads [211] to evaluate in-memory caching systems. Such sources either have different characteristics or do not capture all the characteristics of production in-memory caching workloads. For example, key-value database benchmarks and synthetic workloads don't consider how object size distribution changes over time, which impacts both miss ratio and throughput of in-memory caching systems.

This chapter bridges this gap by collecting and analyzing workload traces from 153 Twemcache [324] clusters at Twitter, one of the most influential social media companies known for its real-time content. Our analysis sheds light on several vital aspects of in-memory key-value cache overlooked in existing studies and identifies areas that need further innovations. The traces used in this paper are made available to the research community at `https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/`.

To the best of our knowledge, this is the first work that studied over 100 different cache workloads covering a wide range of use cases. We believe these workloads are representative of cache usage at social media companies and beyond and hopefully provide a foundation for future key-value cache system designs. Here's a summary of the discoveries:

- In-memory key-value cache does not always serve read-heavy workloads; write-heavy (defined as write ratio $> 30\%$) workloads are very common, occurring in more than 35% of the 153 cache clusters we studied.
- TTL must be considered in the in-memory key-value cache because it limits the effective (unexpired) working set size. Efficiently removing expired objects from the cache needs to be prioritized over cache eviction.
- In-memory key-value cache workloads follow approximate Zipfian popularity distribution, sometimes with very high skew. The workloads that show the most deviations tend to be write-heavy workloads.
- The object size distribution is not static over time. Some workloads show both diurnal patterns and experience sudden, short-lived changes, which pose challenges for slab-based key-value caches such as Memcached.
- Under reasonable cache sizes, FIFO often shows similar performance as LRU, and LRU often exhibits advantages only when the cache size is severely limited.

Chapter 4

Segment-structured Key-Value Cache

The previous chapter discusses the observations in key-value cache workloads, such as small object sizes, wide use of TTLs, and dynamic object size distribution. These patterns are dramatically different from the traditional block caches, which often store uniform-size blocks and are accessed by local applications without going through networks.

This chapter shows that most key-value cache designs do not explicitly optimize for small objects or TTLs, which motivates the design of Segcache, a new storage layout for key-value cache that (1) uses an approximate-TTL index to remove expired objects quickly and (2) segment-structured storage to reduce fragmentation and object metadata overhead.

4.1 Background

As a critical component of the real-time serving infrastructure, caches prefer to store data, especially small objects, in DRAM. DRAM is expensive and energy-hungry. However, existing systems do not use the costly DRAM space efficiently. This inefficiency mainly comes from three places. First, existing solutions are not able to quickly remove expired objects. Second, metadata overhead is considerable compared to typical object sizes. Third, internal or external memory fragmentation is common, leading to wasted space. While improvements of admission [42, 43, 109, 113], prefetching [369], and eviction algorithms [36, 38, 48, 86, 87, 193, 312] have been the main focus of existing works on improving memory efficiency [38, 43, 48, 109], little attention has been paid to addressing

Table 4.1: Comparison of research systems (with corresponding baselines).

System	Memory Allocator	Memory fragmentation	Improve expiration	Object metadata size	Throughput	Memory efficiency improvement approach
MICA	Log	No	No	Decrease	Higher	Worse
Memshare	Log	No	No	Increase	Lower	Memory partitioning and sharing
pRedis	Malloc	External	No	Increase	Lower	Better eviction
Hyperbolic	Malloc	External	No	Increase	Lower	Better eviction
LHD	Slab	Internal	No	Increase	Lower	Better eviction
MemC3	Slab	Internal	No	Decrease	Higher	Small metadata
Segcache	Segment	No	Yes	Minimal	Higher	Holistic redesign

Table 4.2: Techniques for removing expired objects

Technique	Remove all expired?	Is removal cheap?
Deletion on access	No	Yes
Checking LRU tail	No	Yes
Transient item pool	No	Yes
Full cache scan	Yes	No
Random sampling	No	No

expiration and metadata reduction [118, 293]. On the contrary, many systems add more per-object metadata to make smarter decisions about what objects to keep [38, 86, 291, 369].

We summarize recent advancements in in-memory caching systems in Table 4.1 and discuss them below.

4.1.1 TTL and expiration in caching

TTLs are extremely common in caching. As a result, object expiration is an integral part of all existing solutions.

The prevalence of TTL. TTLs are used by users of Memcached and Redis [133, 229, 282, 379], Facebook [41], Netflix [218]. In Twitter's production, *all* in-memory cache workloads use TTLs between one minute and one month. A TTL is specified at write time to determine how long an object should remain accessible in the caching system. An expired object cannot be returned to the client, and a cache miss is served instead.

Cache TTLs serve three purposes. First, clients use TTLs to limit data inconsistency [282, 370]. Writing to the cache is usually best-effort, so it is not uncommon for data in the cache and database to fall out of sync. Second, some services use TTLs to prompt periodic re-computation. For example, a recommendation system may only want to reuse cached results within a time window and recompute periodically to incorporate new activities and content. Third, TTLs are used for implicit deletion. A typical scenario is rate-limiting. Rate limiters are counters associated with some identities. Services often need to cap requests from a particular identity within a predefined time window to prevent denial-of-service attacks. Services store rate limiters in distributed caches with TTLs so that the counts can be shared among stateless services and reset periodically. Another increasingly common scenario is using TTLs to ensure data in caches comply with privacy laws [128, 303].

Lazy expiration. Lazy expiration means expiration only happens when an object is reaccessed. *Deletion on access* is the most straightforward approach adopted by many production caching systems. If a system uses lazy expiration only, an object that's no longer accessed can remain in memory long past expiration.

Proactive expiration. Proactive expiration is used to reclaim memory occupied by expired objects more quickly. Although there has been no academic research on this topic to the best of our knowledge, we identified four approaches introduced into production systems over the years, as summarized in Table 4.2.

Checking LRU tail is used by Memcached. Before eviction is considered, the system checks a fixed number of objects at the tail of the LRU queue and removes expired objects. Operations on object LRU queues reduce thread scalability due to the extensive use of locking for concurrent accesses [38, 48]. Additionally, this approach is still opportunistic and therefore doesn't guarantee the timely removal of expired objects. Many production caches track billions of objects over a few LRU queues, so the time for an object to percolate through the LRU queue is very long.

Transient object pool was introduced by Facebook [259]. It makes a special case for the timely removal of objects with small TTLs. The main idea is to store such objects separately, and only allow them to be removed via expiration. However, choosing the TTL threshold is non-trivial and can have side effects [370]. Although Memcached supports it, it is disabled by default.

Full cache scan is a popular approach adopted by Memcached and CacheLib [41]. As the name indicates, this solution periodically scans all the cached objects to remove expired ones. A full cache scan is very effective if the scan is frequent, but it wastes resources on objects that are not expired, which can be the vast majority.

Random sampling is adopted by Redis. The key idea is to periodically sample a subset of objects and remove expired ones. In Redis, if the percentage of the expired objects in the sample is above a threshold, this process continues. While sampling is cheaper per run, the blind nature of sampling decides that it is both inefficient and not very effective. Users have to accept that the sampling can only keep the percentage of expired objects at a pre-configured threshold. Meanwhile, the cost can be higher than a full cache scan due to random memory access. There have been some production incidents where Redis could not remove enough expired objects and caused unexpected evictions [321].

Despite the various flaws, proactive expiration is highly regarded by developers of production systems. When asked to replace LRU for a better eviction strategy in Memcached, the maintainer states that "*pulling expired items out actively is better than almost any other algorithmic improvement (on eviction) I could think of.*" [229] Meanwhile, Redis' author mentioned that "*Redis 6 expiration will no longer be based on random sampling but will take keys sorted by expiration time in a radix tree.*" [1]

In summary, efficiently and effectively removing expired objects is an urgent problem that needs to be solved in current caching systems.

4.1.2 Object metadata

We observe that the objects stored in in-memory caches are small [370], and the mean object sizes (key+value) of Twitter's top four production clusters are 230, 55, 294, and 72 bytes, respectively. This observation aligns with the observations at Facebook [25].

Existing systems are not efficient in storing small objects because they store a considerable amount of metadata per object. For example, Memcached stores *56 bytes* of metadata

[1] As of Redis v6.0.6, this change is not implemented yet.

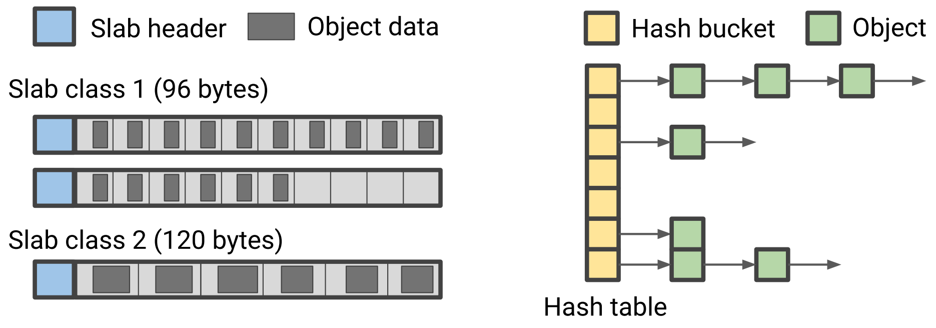


Figure 4.1: Slab memory allocation (left) and object-chained hash table (right) in Memcached.

with each object [2], which is a significant overhead compared to typical object size. All of the metadata fields are critical for Memcached's operations, and cannot be dropped without first removing some functionalities or features.

There have been several attempts at Twitter to cut metadata overhead. For example, Pelikan's slab-based storage removes object LRU queues and reuses one pointer for both the hash chain and free object chain. As a result, it reduces object metadata to 38 bytes. However, this prevents Pelikan from applying the LRU algorithm to object eviction and results in a higher miss ratio compared to Memcached in our evaluation. Pelikan also introduced Cuckoo hashing [266] as a storage module for fixed-size objects, only storing 6 bytes (or 14 bytes with `cas`) of metadata per key.

Several works have also looked at reducing metadata size. RAMCloud [293] and FASTER [70] use a log-structured design to reduce object metadata. However, their designs target *key-value stores* instead of key-value caches (See discussion in section 4.4). MemC3 [118] redesigns the hash table with Cuckoo hashing and removes LRU chain pointers. However, it does not consider some operations such as `cas` for atomic updates, and does not support TTL expiration or other advanced eviction algorithms.

[2] 2×8 bytes LRU pointers, 8 bytes hash pointer, 4 bytes access time, 4 bytes expire time, 8 bytes object size, 8 bytes `cas` (compare-and-set, used for atomic update).

4.1.3 Memory fragmentation

Memory management is one of the fundamental design aspects of an in-memory caching system. Systems that directly use external memory allocators (e.g., *malloc*) such as Redis are vulnerable to external memory fragmentation and OOM.

To avoid this problem, other systems such as Memcached use a slab-based memory allocator, allotting a fixed-size slab at a time, which is then explicitly partitioned into smaller chunks for storing objects, as shown in Figure 4.1 (left). The chunk size is decided by the `class id` of a slab and configured during startup. A slab-based memory allocator is subjected to internal memory fragmentation at the end of each chunk and at the end of each slab.

Using a slab-based allocator also introduces the slab calcification problem, a phenomenon where some slab classes cannot obtain enough memory and exhibit higher miss ratios. Slab calcification happens because slabs are assigned to classes using the first-come-first-serve method. When popularity among slab classes changes over time, the newly popular slab classes cannot secure more memory because all slabs have been assigned. This has been studied in previous works [61, 151, 370]. Memcached automatically migrates slabs between classes to solve this problem, however, it is not always effective [235, 237, 239, 240]. Re-balancing slabs may increase the miss ratio because all objects on the outgoing slab are evicted. Moreover, due to workload diversity and complexity in slab migration, it is prone to errors and sometimes causes crashes in production [236, 238].

Overall, existing production systems have not yet entirely solved the memory fragmentation problem. Among the research systems, log-structured designs such as MICA [199, 203], memshare [87] and RAMCloud [293] do not have this problem. However, they cannot perform proactive expiration and are limited to using basic eviction algorithms (such as FIFO or CLOCK) with low memory efficiency.

4.1.4 Throughput and scalability

In addition to memory efficiency, throughput and thread scalability are also critical for in-memory key-value caches. Memcached's scalability limitation is well documented in various industry benchmarks [242, 259]. The root cause is generally attributed to the

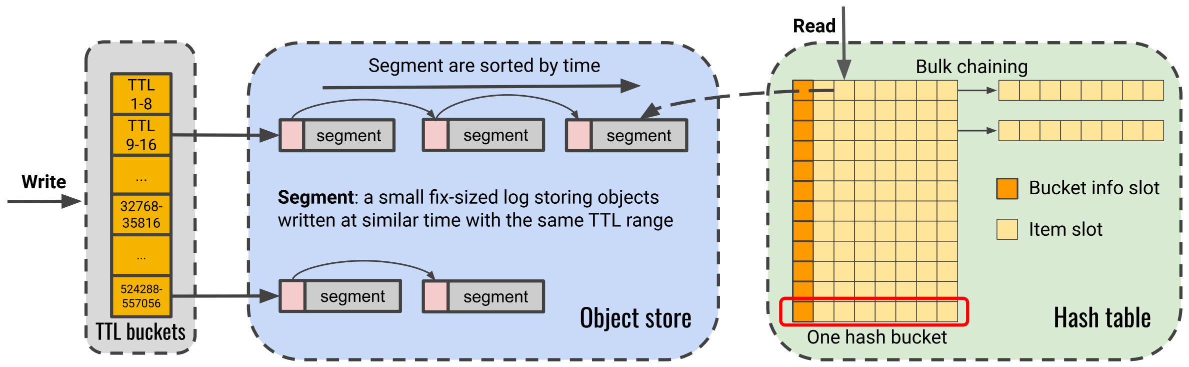


Figure 4.2: Overview of Segcache. A read request starts from the hash table (right), and a write request starts from the TTL buckets (left).

extensive locking in the object LRU queues, free object queues, and the hash table. Several systems have been proposed to solve this problem. Some of them remove locking by using simpler eviction algorithms and sacrificing memory efficiency [118, 203]. Some introduce opportunistic concurrency control [118], which does not work well with write-heavy workloads. Some other works use random eviction algorithms to avoid concurrent reads and writes [38, 48], which do not address all the locking contention. Moreover, they reduce throughput due to the large number of random memory accesses.

4.2 Design principles and overview

The design of Segcache follows three principles.

Be proactive, don't be lazy. Expired objects offer no value, so Segcache eagerly removes them for memory efficiency.

Maximize metadata sharing for economy. To reduce the metadata overhead without loss of functionality, Segcache maximizes metadata sharing across objects.

Perform macro management. Segcache operates on segments to expire/evict objects in bulk with minimum locking.

At a high level, Segcache contains three components: a hash table for object lookup, an object store comprised of segments, and a TTL-indexed bucket array (Figure 4.2).

4.2.1 TTL buckets

Indexing on TTL facilitates the efficient removal of expired objects. To achieve this, Segcache first breaks the spectrum of possible TTLs into ranges. We define the time-width of a TTL range t_1 to t_2 ($t_1 < t_2$) as $t_2 - t_1$. All objects in the range t_1 to t_2 are treated as having TTL t_1, which is the *approximate TTL* of this range. Rounding down guarantees an object can only be expired early, and no object will be served beyond expiration. Objects are grouped into small fix-sized groups called segments (see next section), and all the objects stored in the same segment have the same approximate TTL. Second, Segcache uses an array to index segments based on *approximate TTL*. Each element in this array is called a *TTL bucket*. A segment with a particular approximate TTL value is associated with the corresponding TTL bucket. Within each bucket, segments are chained and sorted by creation time.

To support a wide TTL range from a few seconds to at least one month without introducing too many buckets or losing resolution on the lower end, Segcache uses 1024 TTL buckets, divided into four groups. From one group to the next, the time-width grows by a factor of 16. In other words, Segcache uses increasingly coarser buckets to efficiently cover a wide range of TTLs without losing relative precision for typical TTL buckets. The boundaries of the TTL buckets are chosen in a way that finding the TTL bucket only requires a few bit-wise operations. We show that this design allows Segcache to efficiently and effectively remove expired objects in subsection 4.2.5.

4.2.2 Object store: segments

Segcache uses segments as the basic building blocks for storing objects. All segments are of configurable size, with a default of 1 MB. Unlike slabs in Memcached, Segcache groups objects by approximate TTL, not by size. A segment in Segcache is similar to a small log in log-structured systems. Objects are always appended to the end of a segment, and once written, the objects cannot be updated (except for `incr/decr` atomic operations). However, unlike other log-structured systems [87, 203, 264, 292, 293], where available DRAM is either used as one continuous log or as segments with no relationship between each other, segments in Segcache are sorted by creation time, linked into chains, and indexed by approximate TTLs.

In Segcache, each non-empty TTL bucket stores pointers to the head and tail of a time-sorted segment chain, with the head segment being the oldest. A write request in Segcache first finds the right TTL bucket for the object and then appends to the segment at the tail of the segment chain. When the tail segment is full, a new segment is allocated. If there is no free segment available, eviction is triggered (subsection 4.2.6).

4.2.3 Hash table

As shown in previous works [118], the object-chained hash tables (Figure 4.1 (right)) limits the throughput and scalability in the existing production systems [228, 324]. Segcache uses a bulk-chaining hash table similar to MICA [203] and Faster [70].

An object-chained hash table uses object chaining to resolve hash collisions. The throughput of such a design is sensitive to hash table load. Collision resolution requires walking down the hash chain, incurring multiple random DRAM accesses and string comparisons. Moreover, object chaining imposes a memory overhead of an 8-byte hash pointer per object, which is expensive compared to the small object sizes.

Instead of having just one slot per hash bucket, Segcache allocates 64 bytes of memory (one CPU cache line) as eight slots in each hash bucket (Figure 4.2). The first slot stores the bucket information and the following six slots store object information. The last slot stores either object information or a pointer to the next hash bucket (when more than seven objects hash to the same bucket). This chaining of hash buckets is called bulk chaining. Bulk chaining removes the need to store hash pointers in the object metadata and improves the throughput of hash lookup by minimizing random accesses.

The bucket information slot stores an 8-bit spin lock, an 8-bit slot usage counter, a 16-bit last-access timestamp, and a 32-bit `cas` value. Each item slot stores a 24-bit segment id, a 20-bit offset in the segment, an 8-bit frequency counter, and a 12-bit tag. The tag of a key is a hash used to reduce the number of string comparisons when hash collisions happen.

4.2.4 Object metadata

Segcache achieves low metadata overhead by *sharing metadata across objects*. Segcache facilitates metadata sharing at two places: the hash table bucket and the segment. Objects in the same segment share creation time, TTL, and reference counter, while objects in the same hash bucket share last-access timestamp, spinlock, `cas` value, and hash pointer.

Because objects in the same segment have the same approximate TTL and are written around the same time, Segcache computes the approximate expiration time of the whole segment based on the oldest object in the segment and the approximate TTL of the TTL bucket. This approximation skews the clock and incurs early expiration for objects later in the segment. As we will show in our evaluation, early expiration has a negligible impact on the miss ratio.

Segcache also omits object-level hash chain pointers and LRU chain pointers. Bulk chaining renders the hash chain pointer unnecessary. The LRU chain pointers are not needed because both expiration and eviction are performed at the segment level. Segcache further moves up metadata needed for concurrent accesses (reference counter) into the segment header. In addition, to support `cas`, Segcache maintains a 32-bit `cas` value per hash bucket and shares it between all objects in the hash bucket. While sharing this value may increase false data race between different objects hashed to the same hash bucket, in practice, the impact of this compromise is negligible due to two reasons. First, `cas` traffic is usually orders of magnitude lower than simple read or write, as observed in production environment [370]. Second, one `cas` value is shared only by a few keys, the chance of concurrent updates on different keys in the same hash bucket is small. In the case of a false data race, the client usually retries the request.

The final composition of object metadata in Segcache contains one 8-bit key size, one 24-bit value size, and one 8-bit flag. And Segcache stores only 5 bytes[3] of metadata with each object, which is a 91% reduction compared to Memcached.

4.2.5 Proactive expiration

In Segcache, all objects in one segment are written sequentially and have the same approximate TTL, which makes it feasible to remove expired objects in bulk. Proactive removal of expired objects starts with scanning the TTL buckets. Because segments linked in each TTL bucket are ordered by creation time and share the same approximate TTL, they are also ordered by expiration time. Segcache uses a background thread to scan

[3]The 5-byte does not include the shared metadata, which is small per object. And it also does not include the one-byte frequency counter, which is stored as part of the object pointer in the hash table.

the first segment's header in each non-empty TTL bucket. If the first segment is expired, the background thread removes all the objects in the segment, then continues down the chain until it runs into one segment that is not yet expired, at which point it will move onto the next TTL bucket.

Segcache's proactive expiration technique uses memory bandwidth efficiently. Other than reading the expired objects, each full scan only access *a small amount of consecutive metadata* — the TTL bucket array. This technique also ensures that memory occupied by expired objects is promptly and completely recycled, which improves memory efficiency.

As mentioned before, objects are subject to early expiration. However, objects are usually less useful near the end of their TTL. Our analysis of production traces at Twitter shows that a small TTL reduction makes a negligible difference (if any) in the miss ratio.

4.2.6 Segment eviction

While expiration removes objects that cannot be used in the future and is preferred over eviction, the cache cannot rely on expiration alone. All caching systems support eviction when necessary to make room for new objects.

Eviction decisions can affect the effectiveness of cache in terms of the miss ratio, thus have been the main focus of many previous works [38, 48, 82, 193, 263, 320]. Segcache does not update objects in place. Instead, it appends new objects and marks the old ones as deleted. Therefore, better eviction becomes even more critical.

Unlike most existing systems performing evictions by objects, Segcache performs eviction by segments. Segment eviction could evict popular objects, increasing the miss ratio. To address this problem, Segcache uses a *merge-based* eviction algorithm. The basic idea is that by combining multiple segments into one, Segcache selectively retains a relatively small portion of the objects that are more likely to be accessed again and discards the rest. This design brings out several finer design decisions. First, we need to pick the segments to be merged. Second, there needs to be an algorithm making per-object decisions while going through these segments.

Segment selection. The segments merged during each eviction are always from a single TTL bucket. Within this bucket, Segcache merges the first *N consecutive*, *un-expired*, and *un-merged* (in current iteration) segments (Figure 4.3). The new segment created from

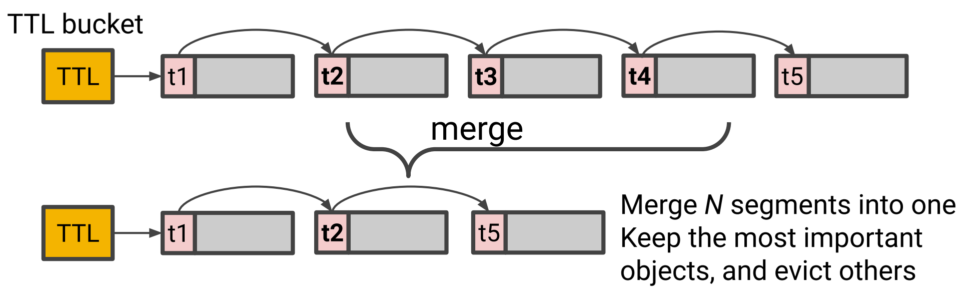


Figure 4.3: Merge-based segment eviction.

the eviction inherits the creation time of the oldest evicted segment. This design has the following benefits. First, the created segment can be inserted in the same position as the evicted segments in the segment chain and maintains the time-sorted segment chain property. Second, objects in the created segment still have relatively close creation/expiration time, and the merge distorts their expiration schedules minimally.

While within one TTL bucket, the segment selection is limited to consecutive ones, across TTL buckets, Segcache uses a round-robin policy to choose TTL buckets.

One-pass merge and segment homogeneity. When merging N consecutive segments into one, Segcache uses a dynamic threshold for retaining objects to implment the merge operation in a single pass. This threshold is updated after scanning every $\frac{1}{10}$ of a segment and aims to retain $\frac{1}{N}$ bytes from each segment being evicted.

The rationale for retaining a similar number of bytes from each segment is that objects and segments created at a similar time are homogeneous with similar properties. Therefore, no segment is more important than others. Figure 4.4a shows the relative standard deviation (RSD, $\frac{std}{mean}$) of the mean object size in consecutive segments and across random segments, and Figure 4.4b compares the RSD of live bytes in consecutive and random segments. Both figures demonstrate that consecutive segments are more homogeneous (similar) than random segments. As a result, retaining a similar number of bytes from each is reasonable. However, we remark that the current segment selection and merge heuristics may not be the optimal solution in some cases, and deserve more exploration.

Selecting objects. So far, one question remains unsolved: what objects should be retained in an eviction? An eviction algorithm's effectiveness is determined by its ability to predict future access based on past information. Under the independent reference model (IRM), a popular model used for cache workloads, an object with a higher frequency is more likely to be re-accessed. And it has been shown to be effective by many works [108, 302].

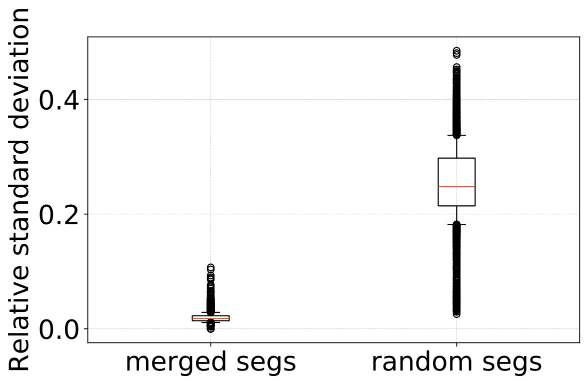
Similar to greedy dual size frequency [82], Segcache uses the frequency-over-size ratio to rank objects. Therefore it needs a frequency counter that is memory-efficient, computationally cheap, and scalable. Meanwhile, it should allow Segcache to be burst-resistant and scan-resistant. Moreover, The counter needs to provide *higher accuracy for less popular* objects (opposite of the counter-min sketch). This is critical for cache eviction because the highly-popular objects are always retained (cached), and the less popular objects decide the miss ratio of a cache. Segcache uses a novel one-byte counter (stored in the hash table), which we call *approximate and smoothed frequency counter* (ASFC), to track frequencies.

Approximate counter. ASFC has two stages. When the frequency is smaller than 16 (the last four bits of the counter), it always increases by one for every request. In the second stage, it counts frequency similar to a Morris counter [346], which increases with a probability that is the inverse of the current value.

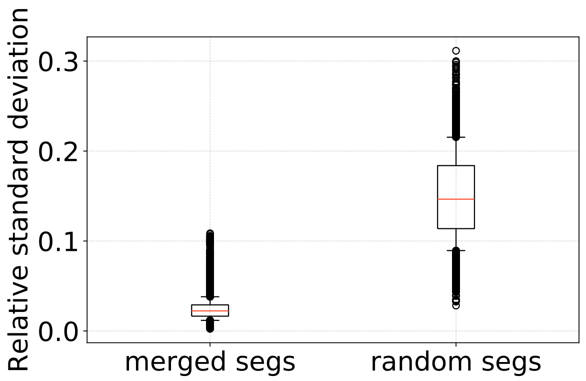
Smoothed counter. Segcache uses the last access timestamp, which is shared by objects in the same hash bucket, to rate-limit updates to the frequency counters. The frequency counter for each object is incremented at most once per second. This technique is effective in absorbing sudden request bursts.

Simple LFU is susceptible to cache pollution due to request bursts and non-constant data access patterns. While several approaches such as dynamic aging [23, 108], and window-based frequency [176] have been proposed to address this issue, they require additional parameters and/or extensive tuning [22]. To avoid extra parameters, Segcache resets the frequency of retained objects during evictions, which has a similar effect as window-based frequency.

The linear increase at low frequency and probabilistic increase at high frequency allow ASFC to achieve higher accuracy for less popular objects. Meanwhile, the approximate design allows ASFC to be memory efficient, using one byte to count up to $2^8 \times 2^8$ requests. The smoothed design of ASFC allows Segcache to be burst-resistant and scalable.



(a) Object size



(b) Live bytes

Figure 4.4: a) The relative standard deviation of mean object size in consecutive segments and random segments. b) Relative standard deviation of live bytes in consecutive segments and random segments.

4.2.7 Thread model and scalability

Segcache is designed to scale linearly with the number of threads by using a combination of techniques such as minimal critical sections, optimistic concurrency control, atomic operations, and thread-local variables. Most notably, because object life cycle management is at the segment level, only modifications to the segment chains require locking, which avoids common contention spots related to object-level bookkeeping, such as maintaining free-object queues. This macro management strategy reduces locking frequency by four orders of magnitude in our default setting compared to what would be needed in a Memcached-like system.

More specifically, no locking is needed on the read path except to increment object frequency, which is at most once every second. On the write path, because segments are append-only, inserting objects can take advantage of atomic operations. However, we observe that relying on the atomic operation is insufficient to achieve near-linear scalability with more than eight threads. To solve this, each thread in Segcache maintains a local view of active segments (the last segment of each segment chain), and the active segments in each thread can be written only by that thread. Although the segments are local to each thread for writes, the objects that have been written are immediately available for reading by other threads. During an eviction, locking is required when segments are being removed from the segment chain. However, the critical section of removing a segment from the chain is very tiny compared to object removal, which is *lock-free*. Moreover, evicting one segment means evicting thousands of objects, so segment eviction is infrequent compared to object writes.

4.3 Implementation and Evaluation

Table 4.3: Traces used in evaluation

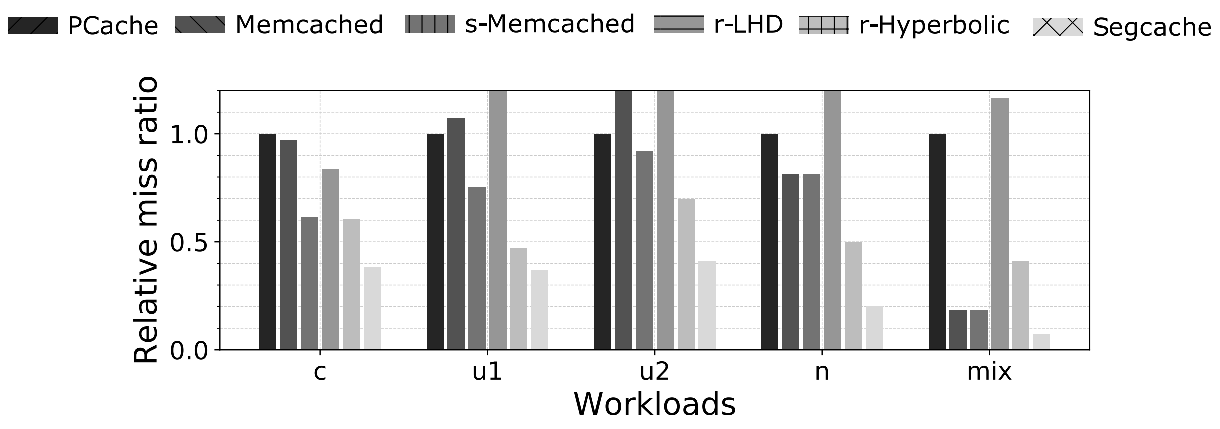
Trace	Workload type	# requests	TTLs (TTL: percentage)	Write ratio	Mean object size	production miss ratio
c	content	4.2 billion	1d: 65%, 14d: 27%, 12h: 7%	7%	230 bytes	1-5%
$u1$	user	6.5 billion	5d:1.00	1%	290 bytes	<1%
$u2$	user	4.5 billion	12h:1.00	3%	55 bytes	<1%
n	negative cache	1.6 billion	30d:1.00	2%	45 bytes	∼1%
mix	content + user + negative cache + transient item	12 billion	30d: 14%, 14d:11%, 24h: 23%, 12h: 38%, 2min:12%	7%	243 bytes	NA

In this section, we compare the memory efficiency, throughput, and scalability of Segcache against several research and production solutions, using traces from Twitter's production. Specifically, we are interested in the following questions,

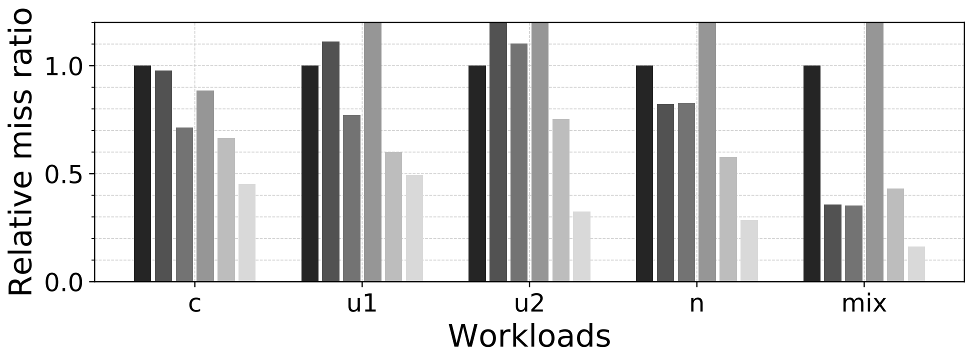
- Is Segcache more memory efficient than alternatives?
- Does Segcache provide comparable throughput to state-of-the-art solutions? Does it scale well with more cores?
- Is Segcache sensitive to design parameters? Are they easy to pick or tune?

4.3.1 Implementation

Segcache is implemented as a storage module in the open-sourced Pelikan project. Pelikan is a cache framework developed at Twitter. The Segcache module can both work as a library or be setup as a Memcached-like server. Our current implementation supports multiple worker threads, with a dedicated background thread performing proactive expiration. For our evaluation, eviction is performed by worker threads as needed, but it



(**a**) Large cache size



(**b**) Small cache size

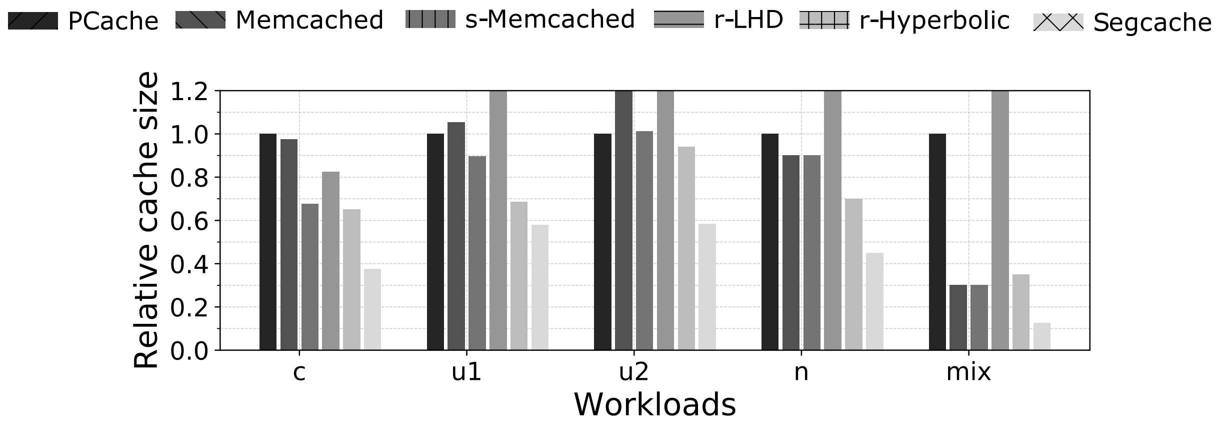
Figure 4.5: Relative miss ratio of different systems (baseline Pelikan is 1), lower is better.

is easy to use the same background thread to facilitate background eviction. We provide configurable options to change the number of segments to merge for eviction and segment size. The source code can be accessed at `http://www.github.com/pelikan-io/pelikan` and archived at `http://www.github.com/thesys-lab/segcache`.

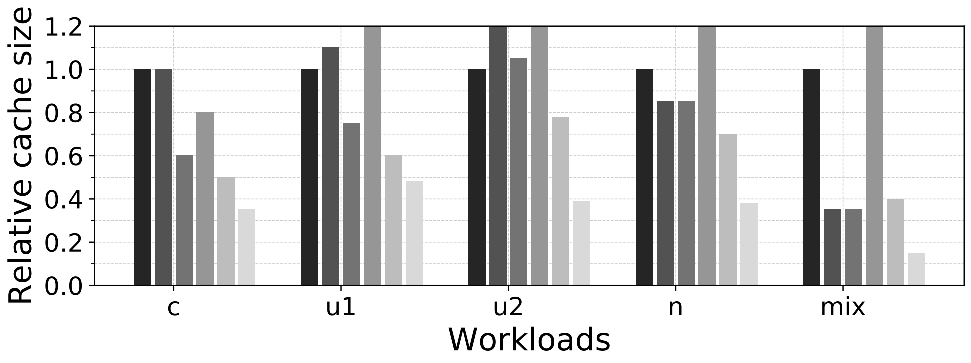
4.3.2 Experiment setup

Traces

We used week-long unsampled traces from production cache clusters at Twitter (Table 4.3, the same as in previous work [370]). Trace c comes from a cache storing tweets and their metadata, which is the largest cache cluster at Twitter. Trace $u1$ and $u2$ are both user related, but the access patterns of the two workloads are different, so different TTLs



(**a**) Production miss ratio



(**b**) Miss ratio of Pelikan at the small cache size in Figure 4.5b

Figure 4.6: Relative memory footprint to achieve a certain miss ratio, lower is better.

are used. Notably, they are separated into two caches in production because effective and efficient proactive expiration was not achievable prior to Segcache. Trace n is a negative result cache, which stores the keys that do not exist in the database, a common way of using cache to shield databases from unnecessary high loads.

Although Twitter's production deployments are single-tenant, multi-tenant deployments are also common because of better resource utilization [25]. To evaluate the performance under multi-tenant workloads, we merged workloads from four types of caches: user, content, negative cache, and transient item cache.

Baselines

Memcached used in our evaluation is version 1.6.6 with segmented LRUs. It supports lazy expiration and checks the LRU tail for expiration. We ran Memcached in two modes, one with cache scanning enabled (s-Memcached), which scans the entire cache periodically to remove expired objects; the other with scanning disabled (Memcached). Other expiration techniques are enabled in both modes. Our evaluation also includes pelikan_twemcache (PCache), Twitter's Memcached equivalent, and successor to Twemcache [12]. Compared to Memcached, PCache has smaller object metadata without LRU queues and only performs slab eviction [381]. We implemented LHD [38] and Hyperbolic [48] on top of PCache since original implementations are not publicly available. These systems do not consider object expiration. To make the comparisons fairer, we add random sampling to remove expired objects in these two systems, which is also how Redis performs expiration. In the following sections, r-LHD and r-Hyperbolic refer to these enhanced versions. Note that adding random sampling to remove expired objects does not significantly impact the throughput, and we observe less than a 10% difference.

Because we do not modify the networking stack, we focus our evaluation on the storage subsystem. We performed all evaluations by close-loop trace replay on dedicated hosts in Twitter's production fleet. The hosts have dual-socket Intel Xeon Gold 6230R CPU, 384 GB DRAM with one 100 Gbps NIC.

Metrics

We use three metrics in our evaluation to measure the memory efficiency, throughput, and scalability of the systems.

Relative miss ratio. Miss ratio is the most common metric in evaluating memory efficiency. Because workloads have dramatically different miss ratios in production (from a few percent to less than 0.1%) and compulsory miss ratios, directly plotting miss ratio is less readable. Therefore, we use *relative miss ratio* (defined as $\frac{mr}{mr_{baseline}}$ where mr stands for miss ratio and the baseline is PCache) in the presentation.

Relative memory footprint. Although miss ratio is a common metric, a sometimes more useful metric is how much memory footprint can be reduced at a certain miss ratio. Therefore, in subsection 4.3.3, we show this metric using PCache memory footprint as the baseline.

Throughput and scalability. Throughput is measured in million queries per second (MQPS) and used to quantify a caching system's performance. Scalability measures the throughput running on a multi-core machine with the number of hardware threads from 1 to 24 in our evaluations.

4.3.3 Memory efficiency

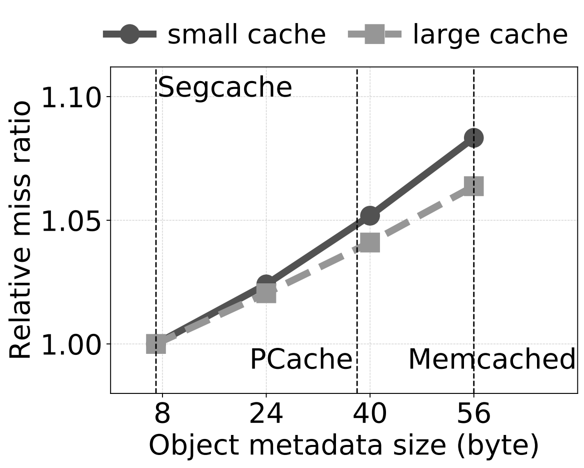
In this section, we compare the memory efficiency of all systems. We present the relative miss ratio at two cache sizes[4] in Figure 4.5. (1) The "large cache" is the cache size when the miss ratio of Segcache reaches the plateau (<0.05% miss ratio reduction when the cache size increases by 5%). Miss ratios achieved at large cache sizes are similar to production miss ratios. (2) we choose the "small cache" size as 50% of the large cache size.

Compared to the best of the five alternative systems, Segcache reduces miss ratios by up to 58%. Moreover, it performs better on both the single-tenant and the multi-tenant workloads. This large improvement is the cumulative effect of having timely proactive expiration, small object metadata, no memory fragmentation, and a merge-based eviction strategy.

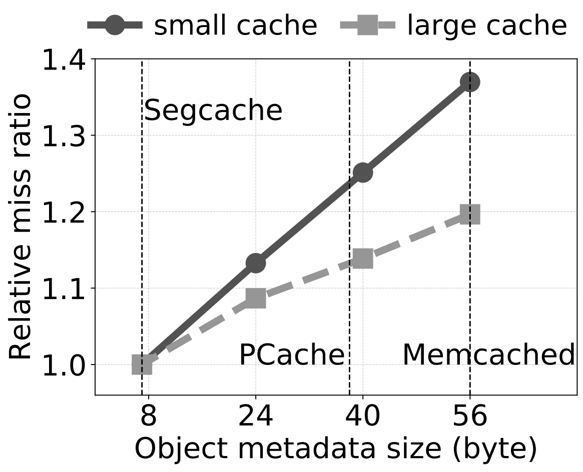
We observe that Memcached and PCache have comparable miss ratios in most workloads (except workload mix because PCache is not designed for multi-tenant workloads). While comparing Memcached and s-Memcached, we observe that adding full cache scanning capability significantly reduces the miss ratio by up to 40%, which indicates the importance of proactive expiration. However, as we show in subsubsection 4.3.4, cache scanning is expensive and reduces throughput by almost half for some workloads. Moreover, we observe that workload n and mix do not benefit from full cache scanning. Workload n shows no benefit because it uses a single TTL of 30 days and no objects expire in the evaluation. Although workload mix has a mixture of short and long TTLs, it shows no benefit because the objects of different TTLs are from different workloads with different object sizes, and are stored in different slab classes with different LRU queues. As a result, checking LRU tail for expiration is effective at removing expired objects and scanning provides little benefit. Overall, we observe that proactively removing expired objects can effectively reduce miss ratio and improve memory efficiency.

State-of-the-art research caching systems, r-LHD and r-Hyperbolic use ranking to select eviction candidates and often reduce miss ratio compared to LRU. In our evaluation, r-Hyperbolic shows lower miss ratio compared to Memcached and PCache, while r-LHD is only better on workload c. r-LHD is designed for workloads with a mixture of scan and LRU access patterns (such as block access in storage systems), while in-memory caching workloads rarely show scan requests. This explains why it has higher miss ratios.

[4]We experimented with twenty cache sizes, and the two set of results presented here are representative.



(a) Workload c



(b) Workload n

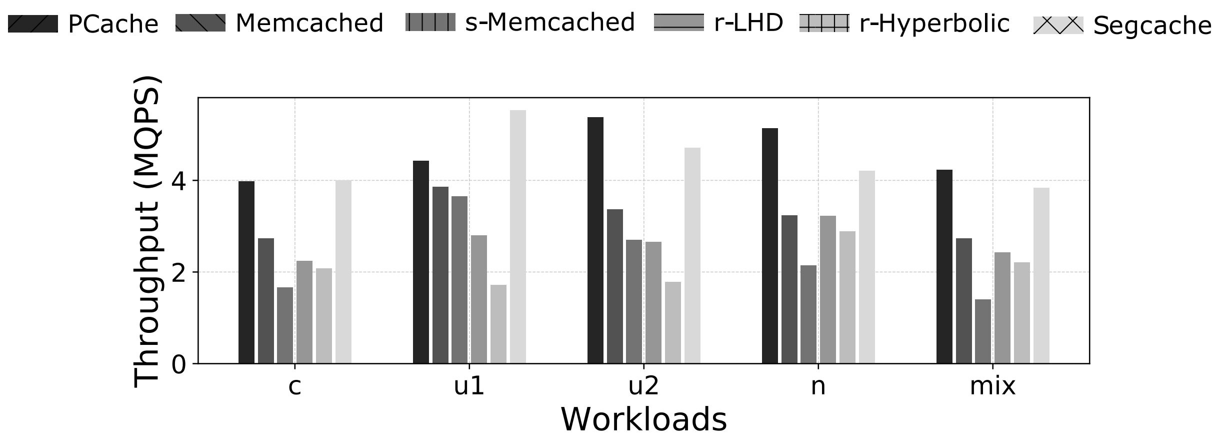
Figure 4.7: Impact of object metadata size on miss ratio. Workload n has smaller object sizes as compared to workload c and hence enjoys larger benefits from reduction from object metadata.

We have also evaluated r-LHD and r-Hyperbolic without sampling for expiration (not shown), and as expected, they have higher miss ratios due to the wasted cache space from expired objects.

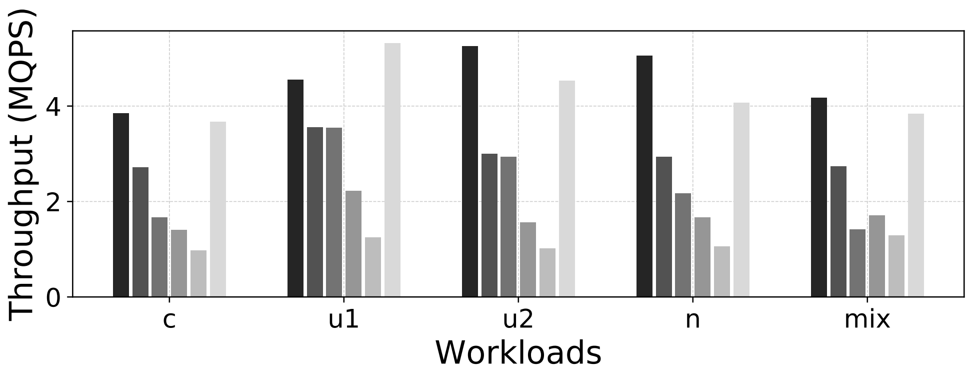
An alternative way of looking at memory efficiency is to determine the cache size required to achieve a certain miss ratio. We show the relative memory footprints of different systems in Figure 4.6, using PCache as the baseline. The figures show that for both the production miss ratio and a higher miss ratio, Segcache reduces memory footprint by up to 88% compared to PCache, 60% compared to Memcached, 56% compared to s-Memcached, and 64% compared to r-Hyperbolic.

Ablation study

In Figure 4.5, we observe that s-Memcached reduces miss ratio by up to 35% compared to Memcached, which demonstrates the importance of proactive expiration, one of the key design features of Segcache. Besides proactive expiration, another advantage of Segcache over previous systems is smaller object metadata. To understand its impact, we measure the relative miss ratio of increasing object metadata in Segcache (Figure 4.7). It shows that reducing object metadata size can have a large miss raito impact for workloads with small object sizes. Workload c has relatively large object sizes (230 bytes), and reducing the metadata from 56 bytes to 8 bytes reduces the miss ratio by 6-8%. While



(**a**) Large cache size



(**b**) Small cache size

Figure 4.8: Throughput of different systems, the higher the better.

workload n has small object sizes (45 bytes) and reducing object metadata size provides a 20-38% reduction in miss ratio. This result indicates reducing object metadata size is very important, and it is a critical component contributing to Segcache's high memory efficiency.

4.3.4 Throughput and scalability

Single-thread throughput

Besides memory efficiency, the other important metric of a cache is the throughput. Figure 4.8 shows the throughput of different systems. Compared to other systems, PCache and Segcache achieve higher throughput, up to 2.5× faster than s-Memcached, up to

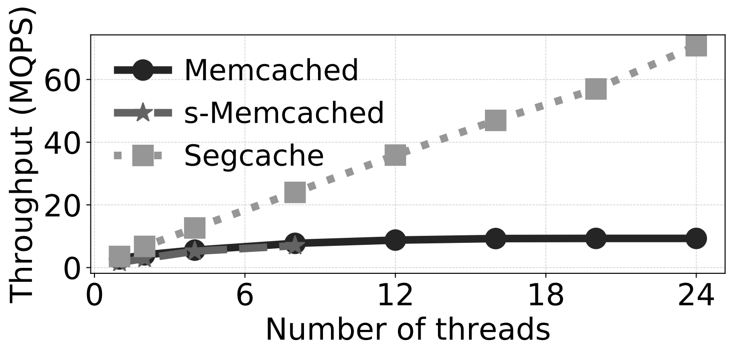
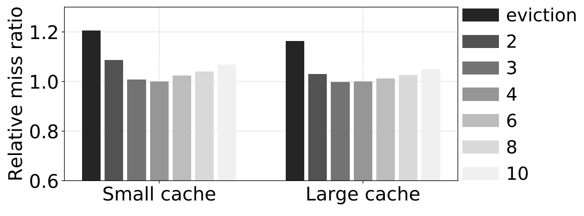
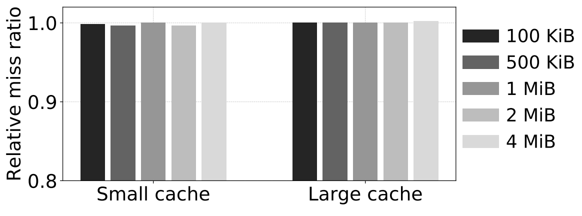


Figure 4.9: Scalability



(**a**) Number of segments to merge



(**b**) Segment size

Figure 4.10: Sensitivity analysis.

3× faster than r-Hyperbolic, and up to 4× faster than r-LHD. The reason is that PCache performs slab eviction only, and Segcache performs merge-based segment eviction. Both systems perform batched and sequential bookkeeping for evictions, which significantly reduces the number of random memory accesses and makes good use of the CPU cache. In addition, PCache and Segcache do not maintain an object LRU chain, which leads to less bookkeeping and also contributes to the high throughput.

Although r-LHD and r-Hyperbolic have lower miss ratios than Memcached, their throughput is also lower. The reason is that both systems use random sampling during evictions, which causes a large number of random memory accesses. One major bottleneck of a high-throughput cache is the poor CPU cache hit ratio, and optimizing CPU cache utilization has been one focus of improving the throughput [203, 220]. Although r-LHD proposes to segregate object metadata for better locality [38], it requires adding more object metadata, and hence would further decrease memory efficiency.

Thread scalability

We show the scalability results in Figure 4.9, where we compare Segcache with Memcached and s-Memcached. Figure 4.9 shows that compared to Memcached, Segcache has a higher throughput and close-to-linear scalability. With 24 threads, Segcache achieves over 70 MQPS, a $19.9\times$ boost compared to using a single-thread, while Memcached only achieves 9 MQPS, $3.4\times$ of its single-thread throughput. The reason why Segcache can achieve close-to-linear scalability is the effect of multiple factors as discussed in subsection 4.2.7. While there is not much throughput difference between Memcached and s-Memcached, s-Memcached is deadlocked when running with more than 8 threads.

Note that we do not present the result of PCache in this figure because it does not support multi-threading. We also do not show the result of r-LHD and r-Hyperbolic because we could not find any simple way to implement a better locking than the one in Memcached. Although r-LHD and r-Hyperbolic removes the object LRU chain and lock, the slab memory allocator still requires heavy locking.

4.3.5 Sensitivity

In this section, we study the effects of parameters in Segcache using workload c (from Twitter's largest cache cluster). The most crucial parameter in Segcache is the number of segments to merge for eviction, which balances between processing overhead and memory efficiency. Figure 4.10a shows how the miss ratio is affected by the number of merged segments. Compared to retaining no objects (the bar labeled eviction), using merge-based eviction reduces the miss ratio by up to 20%, indicating the effectiveness of merge-based eviction. Moreover, it shows that the point for the minimal miss ratio is between 3 and 4. Merging two segments or more than four segments increases the miss ratio, but not significantly.

There are two reasons why merging too few segments leads to a high miss ratio. First, merging too few segments can lead to unfilled segment space. For example, when merging only two segments, 50% of the bytes are retained from each segment in one pass. If the second segment does not have enough live objects, the new segment will have space wasted. Second, the fidelity of predicting future accesses on unpopular objects is low. Merging fewer segments means retaining more objects, so it requires distinguishing unpopular objects, and the decision can be inaccurate. Meanwhile, merging fewer segments means triggering eviction more frequently, giving objects less time to accumulate hits.

On the other hand, merging too many segments increases the miss ratio as well. Because merging more segments means setting a higher bar for retained objects, some important objects can be evicted. In our evaluation, we observe three and four are, in general, good options. However, merging more or fewer segments does not adversely affect the miss ratio significantly and still provides a lower miss ratio than current production systems. Therefore, we consider this parameter a stable one that does not require tuning per workload.

Besides the number of segments to merge, another parameter in Segcache is the segment size. We use the default 1 MB in our evaluation; Figure 4.10b shows the impact of different segment sizes. It demonstrates that segment size has little impact on the miss ratio, which is expected. Because the fraction of objects retained from each segment does not depend on the segment size, thus not affecting the miss ratio.

4.4 Discussion

4.4.1 Alternative proactive expiration designs

Besides the TTL bucket design in Segcache, there are other possible solutions for proactive expiration. For example, a radix tree or a hierarchical timing wheel can track object expiration time. However, neither is as memory efficient as Segcache. In fact, any design that builds an expiration index strictly at the object level requires two pointers per object, an overhead with demonstrated impact for our target workloads. The radix tree may also use an unbounded amount of memory to store the large and uncertain number of expiration timestamps. In addition, performing object-level expiration and eviction requires more random memory access and locking than bulk operations, limiting throughput and scalability.

4.4.2 In-memory key-value cache vs store

In the literature, we observe several instances where there is a mix-up of *volatile key-value caches* (such as Memcached) and *durable key-value stores* (such as RAMCloud and RocksDB). However, from our viewpoint, these two types of systems are significantly different in terms of their usage, requirements, and design. Indeed, one of the main contributions is to identify the opportunity to approximate object metadata and share them (time, pointers, reference counters, version/cas number) across objects. Time approximation in particular is not as tolerated in a traditional key-value store. Below we discuss the differences between caches and stores.

TTL. TTLs are far more ubiquitous in caching than in key-value store [41, 133, 218, 259, 321]. We described Twitter's use of TTLs in detail in [370]. In comparison, many datasets are kept in key-value stores indefinitely.

Eviction. Eviction is unique to caching. In addition, eviction is extremely common in caching. A production cache running at 1M QPS with 10% writes, which can be new objects or on-demand fill from cache misses, will evict 100K objects every second. Repurposing compaction and cleaning techniques in log-structured storage may not be able to keep up with the write rate needed in caching. On the other hand, caches have considerable latitude in deciding what to store and can choose more efficient mechanisms.

Design requirements. In-memory caches are often used in front of key-value stores to absorb most read requests or to store transient data with high write rates. Production users expect caches to deliver much higher throughput and/or much lower tail latencies. In contrast, key-value stores are often considered sources of truth. As such they prioritize durability (crash recovery) and consistency over latency and throughput.

The differences between cache and store allow us to make some design choices in Segcache that are not feasible for durable key-value stores (even if they are in memory).

4.5 Related work

4.5.1 Memory efficiency and throughput

Approaches for improving memory efficiency fall into two categories: improving eviction algorithms and adding admission algorithms.

Eviction algorithm. A vast number of eviction algorithms have been proposed in different areas starting from the early 90s [166, 227, 263, 397]. However, most focus on the cache replacement of databases or page cache, which are different from a distributed in-memory cache because cached contents in databases and page cache are typically fixed-size blocks with spatial locality. In recent years, several algorithms have been proposed to improve the efficiency of in-memory caching, such as LHD [38], Hyperbolic caching [48], pRedis [267], and mPart [60]. However, all of them add more object metadata and computation, which reduces usable cache size and reduces throughput, which has significant repercussions for caches with small objects.

Segcache uses a merge-based eviction strategy that retains high-frequency small-sized objects from evicted segments, which is similar to a frequency-based eviction algorithm such as LFU [23, 176] and GDSF [82]. However, unlike some of these systems that require parameter tuning, Segcache uses ASFC, which avoids these problems. In addition to the eviction algorithm, two major components contributing to Segcache's low miss ratio are efficient, proactive expiration, and object metadata sharing, which are unique to Segcache.

Admission control. Adding admission control to decide which object should be inserted into the cache is a popular approach for improving efficiency. For example, Adaptsize [43], W-TinyLFU [108, 109], flashshield [113] are designed in the recent years. Admission control is effective for CDN caches, which usually have high one-hit-wonder ratios (up to 30%) with a wide range of object sizes (100s of bytes to 10s of GB). Segcache does not employ an admission algorithm because most of the in-memory cache workloads have low one-hit-wonder ratios (<5%) and relatively small object size ranges. Moreover, adding admission control often adds more metadata and extra computation, hurting efficiency and throughput.

Other approaches. There are several other approaches to improving efficiency, such as optimizing slab migration strategy in Memcached [61, 151], compressing cached data [357], and prefetching data [369]. Reducing object metadata size has also been considered in previous works [118]. However, for supporting the same set of functions (including expirations, deletions, `cas`), these approaches need more than twice as much object metadata as Segcache.

Throughput and scalability. A large fraction of works on improving throughput and scalability focus on durable key-value stores [61, 199, 220], which are different from key-value caches as discussed in section 4.4. Segcache is inspired by these works and further improves throughput and scalability by macro management using approximate and shared object metadata.

4.5.2 Log-structured designs

Segcache's segment-structured design is inspired by several existing works that employ log-structured design [61, 77, 203, 264, 292, 293] in storage and caching systems. The log-structured design has been widely adopted in storage systems to reduce random access and improve throughput. For example, log-structured file systems [292] and LSM-tree databases [288] transform random disk writes to sequential writes. Recently, log-structured designs have also been adopted in in-memory key-value store [61, 77, 264, 293] to improve both throughput and scalability.

For in-memory caching, MICA [203] uses DRAM as one big log to improve throughput, but it uses FIFO for eviction and does not optimize for TTL expiration. Memshare [87] also uses a log-structured design and has the concept of segments. However, Memshare optimizes for multi-tenant cache by moving cache space between tenants to minimize miss ratio based on each tenant's miss ratio curve. Memshare uses a cleaning process to scan N segments, evict one segment, and keep $N-1$ segments where the goal is to enforce memory partitioning between tenants. In terms of performance, scanning N ($N = 100$ in evaluation) segments and *evicting one* incurs a high computation overhead and negatively affects the throughput. Moreover, to compute the miss ratio of different tenants, Memshare adds more metadata to the system, which reduces memory efficiency.

Systems employing a log-structured design benefit from reduced metadata size and memory fragmentation and increased write throughput, for example, several of the existing works [118, 288, 293] and including Segcache. Compared to these existing works, Segcache achieves a higher memory efficiency by *approximating* and *sharing* object metadata, *proactive TTL expiration*, and using ASFC to retain fewer bytes during eviction while providing a low miss ratio (10% - 25% bytes from each segment are retained in Segcache compared to 75% in RAMCloud [293] and 99% in Memshare [87]).

In a broad view, Segcache can be described as a *dynamically-partitioned* and *approximate-TTL-indexed* log-structured cache. However, one of the key differences between Segcache and log-structured design is that Segcache is centered around the indexed and sorted segment chain. Both objects in a segment and segments in the chains are time-sorted and indexed by approximate TTLs for metadata sharing, macro management, and efficient TTL expiration.

4.6 Chapter Summary

Achieving high memory efficiency, high throughput, and high scalability simultaneously in caching systems is challenging. This chapter presents Segcache, a new cache storage design that achieves all three desired properties. Segcache is a TTL-indexed, dynamically partitioned, segment-structured cache in which objects of similar TTLs are stored in a small, fixed-size log called a segment. Segments are first grouped by TTL and then naturally sorted by creation time. This design makes the timely removal of expired objects both simple and cheap. As a cache, Segcache performs eviction by merging a few segments into one, retaining only the most important objects, and freeing the rest. Managing the object life cycles at the segment level allows most metadata to be shared within a segment and metadata bookkeeping to be performed with a limited number of tiny critical sections. These decisions improve memory efficiency and scalability without sacrificing throughput or features.

Below are some highlights of our contributions:

- To the best of our knowledge, Segcache is the first cache design that can efficiently remove all objects immediately after expiration. This is achieved through TTL-indexed, time-sorted segment chains.
- This chapter proposed and demonstrated the "object sharing economy", a concept that reduces per-object metadata to just 5 bytes per object, a 91% reduction compared to Memcached, without compromising on features.
- The single pass, merge-based eviction algorithm uses an approximate and smoothed frequency counter to balance between retaining high-value objects and effectively reclaiming memory.
- This chapter introduced a "macro management strategy" in managing a cache system. It shows that replacing per-request bookkeeping with batched operations on segments can significantly improve throughput and provide close-to-linear thread scalability.
- This chapter evaluated Segcache using a wide variety of production traces and compared results with multiple state-of-the-art designs. Segcache reduces memory footprint by 42-88% compared to Twitter's production system and 22-58% compared to the best of state-of-the-art designs.

Chapter 5

Coded Content Delivery Network Cache

Besides key-value caches deployed inside cloud data centers, there is another important category of key-value cache [1] — Content Delivery Networks cache, which is often deployed at the edge of the Internet. CDN operators deploy edge clusters in ISP and IXP close to end users so that they can cache and deliver popular content to users quickly and cheaply. CDNs are one of the foundations of today's Internet because most of the content does not need to be transferred over the unreliable Internet.

Because of the importance of CDNs, having a reliable service is paramount. This chapter looks into the design of reliable Content Delivery Network caches with a focus on reliability within each cluster. This chapter will demonstrate how to use erasure coding to reduce redundancy overhead for fault tolerance. However, naive use of erasure coding does not work in distributed caches because each cache performs evictions independently. This causes the chunks on different servers to be evicted at different times. When a server becomes unavailable, the chunks needed to recover the original objects may have already been evicted from the cluster. To solve the key problem in reliable distributed caching, this chapter introduces parity rebalance, a mechanism that will align the write rates and eviction rates on different servers, allowing servers to perform "synchronized" evictions without coordination.

[1] In some works, a CDN cache is also called an object cache. The main difference between a key-value cache and an object cache in those works is the object size — a cache serving workloads with small objects is called a key-value cache, and a cache serving large objects is called an object cache. However, the size boundary is often arbitrary. Because all the object caches also have a key-value interface, we refer to a cache using a key-value interface as a key-value cache, including the CDN cache.

5.1 Background

CDN Architecture. A CDN is a large distributed system with hundreds of thousands of servers deployed around the world [99, 260]. The servers are grouped into *clusters*, where each cluster is deployed within a data center on the edge of the Internet. The CDN cluster caches content and serves it on behalf of *content providers*, such as e-commerce sites, entertainment portals, social networks, news sites, media providers, etc. By caching content in server clusters proximal to the end users, a CDN improves performance by providing faster download times for clients. Unlike storage systems, CDN servers do not store the original content copies. When the requested content is not available in the cluster (cache miss), the content is retrieved from other CDN clusters or the origin servers operated by the content provider.

Bucket-based request routing. When a user requests an object, such as a web page or video, the *global load balancer* of the CDN routes the request to a cluster that is proximal to the user [75]. Next, the *local load balancer* within the cluster routes the request to one or more servers within the chosen cluster that can serve the requested object. As an example, in Akamai's CDN, these routing steps are performed as DNS lookups. A content provider CNAMEs its domain name (e.g., for all of its media objects) to a sub-domain whose authoritative DNS server is the CDN's global load balancer. At the global load balancer, this sub-domain is CNAME'd to a cluster-local load balancer that assigns the sub-domain to a cluster server using consistent hashing [219].

CDN request routing stands in contrast to sharding in key-value caches, such as Memcached and Redis, where consistent hashing is often applied at a per-object granularity [259, 370]. *In CDNs, load balancing decisions are taken on the granularity of groups of objects called buckets*. Each bucket, in a DNS-based load balancer, corresponds to a domain name that is resolved to obtain one or more server IPs that host objects in that bucket. This resolution is computed using consistent hashing. Since the number of buckets is limited in the range of 100s, the computation is performed and cached when a cluster server becomes available or unavailable.

CDN Performance Requirements. A CDN aims to serve content faster than a customer's origin by a specified speedup factor. This factor is commonly part of a service level agreement (SLA) between the CDN and the content provider. The SLA is monitored by recording download times from a globally distributed set of locations for the same content using both CDN and origin servers. Hence, the goal is to ensure good “tail” performance in *every* time interval for *every* content provider from *every* cluster.

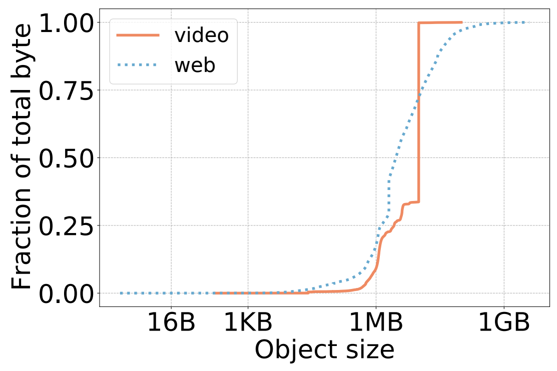
Operating Costs of a CDN. CDNs seek to minimize the operating cost, which consists of the following main categories. (i) *Bandwidth*: A major component of the operating cost of a CDN is bandwidth, accounting for roughly 25% of operating costs. The bandwidth cost can be further broken down, the bandwidth cost caused by cache miss traffic called *midgress* [318] that accounts for roughly 20%, while the rest is the cost of *egress* i.e., the traffic from the CDN servers to clients. CDNs have a great cost incentive to reduce the byte miss ratio and the midgress traffic since a CDN gets paid by content providers for the traffic to end users. The midgress traffic between CDN clusters and the origin is purely a cost overhead for the CDN. Even modest reductions in midgress translate into large cost savings since the bandwidth costs tens of millions of dollars per year for a large CDN[318].

(ii) SSD wearout: A second major cost component is server deprecation which accounts for about 25% of the operating cost of a large CDN. Hardware replacements are particularly expensive for small edge clusters due to the large geographic footprints of CDNs. SSDs are a key component due to the high IOPS requirements of CDN caching. Unfortunately, using SSDs in caching applications is challenging due to their limited write endurance [41, 113, 305, 320]. With deployments of TLC and QLC SSDs, reducing SSD write rates has become even more critical. Besides reducing the average write rate within a cluster, CDNs also seek to reduce the variance of write rates of different servers and their SSDs. Large variance leads to some SSDs not achieving their intended lifetime (e.g., 3 years) as well as high tail latency (see subsection 5.2.4). Consequently, CDNs seek to reduce the peak write rate, ideally balancing write rates across all SSDs in a cluster.

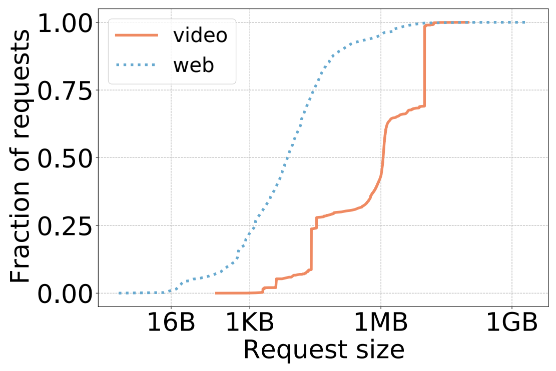
5.2 Production CDN Trace Analysis

This section motivates the design of C2DN by analyzing three sets of traces from production Akamai clusters.

We collected request traces from two typical Akamai 10-server clusters (cluster cache size 40 TB), one mainly serving web traffic and the other mainly serving video traffic. These traces comprise anonymized loglines for every request from every server over a



(**a**) Object sizes



(**b**) Request sizes

Figure 5.1: a) Size distributions show that large objects contribute to most of the unique bytes and b) most requests are for small objects.

Table 5.1: Read and write load for a 10-server production cluster.

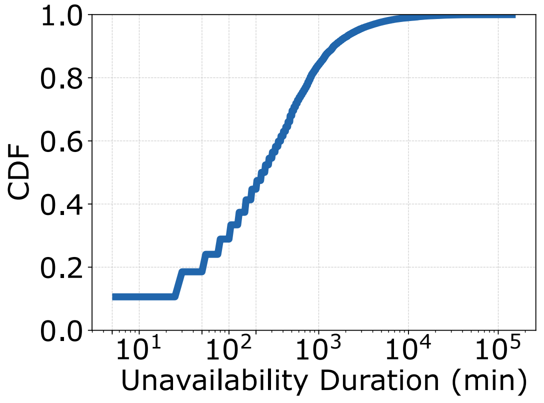
Per-server load (TB)	Max	Min	Mean	Max/min
Weekly read	225.2	167.9	191.2	1.3
Weekly write	16.54	6.69	12.57	2.5

period of 7 and 18 days, respectively. The **web trace** totals 6 billion requests (1.7 PB) for 273 million unique objects (79.8 TB). The **video trace** totals 600 million requests (2.1 PB) for 130 million unique objects (224 TB).

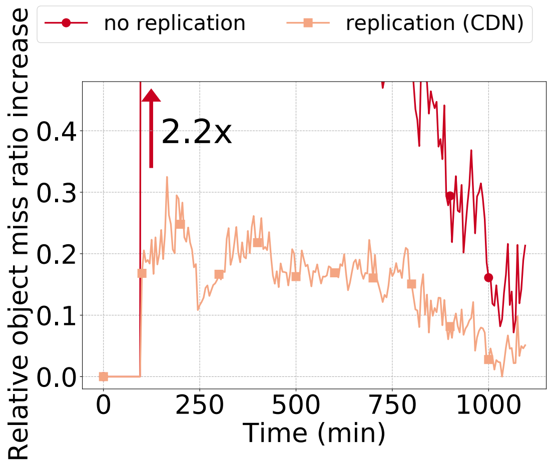
Additionally, we collected availability traces from 2190 Akamai clusters over 31 days. The trace consists of snapshots taken every 5 minutes from each cluster's local load balancer. Each snapshot contains the number of available servers as determined by the load balancer. The smallest cluster has two servers, the largest cluster has over 500 servers, and the median cluster size is 17 servers. We observe that cluster size has a wide range, and around 40% of clusters have fewer than or equal to 10 servers.

5.2.1 Diversity in workloads and object sizes

CDNs mix different types of traffic in clusters to fully use their resources. For example, different "classes" of traffic with small and large object sizes, such as web assets and video-on-demand, are mixed to balance the utilization of the cluster's CPUs as well as network



(a) Unavailability durations



(b) Unavailability impact

Figure 5.2: a) Server unavailabilities are mostly transient. b) Object miss ratios spike after server unavailability both with and without state-of-the-art replication.

and disk bandwidth [317]. Consequently, object sizes vary widely [43]. Figure 5.1a and Figure 5.1b show the size distribution for our production traces, weighted by unique objects and by request count, respectively. As expected, object sizes vary from a few bytes to a few GBs. Fig 5.1a shows that *the majority of traffic and cache space is used by large objects*. Furthermore, objects smaller than 1 MB make up less than 15% and 12% of the total working set in web and video, respectively. Fig 5.1b shows that *the majority of the requests are for small objects* with 95% of requests in web-dominant workload smaller than 1 MB, and 50% of requests in video-dominant workload smaller than 1 MB.

5.2.2 Unavailability is common and transient

Unavailability is common. Across all clusters, server unavailabilities occur in 45.2% of the 5-minute snapshots. For clusters with only ten servers (the same size as the cluster we collect request traces from), we observe that 30.5% of 5-minute time snapshots show server unavailability. Moreover, we observe that unavailability affects only a small number of servers at any given time: 85% of unavailabilities affect less than 10% of servers in large clusters, and 84% of unavailabilities affect no more than a single server in a ten-server cluster. These unavailability rates can appear high compared to published failure rates in large data centers [123, 243, 278, 280, 295] and HPC-systems [301]. However, environmental conditions can be more challenging in small edge clusters. For example, edge locations often have less efficient cooling systems than highly optimized hyper-scale data centers; edge clusters also have less power redundancy, such as redundant battery and generator backups [260]. Moreover, CDN clusters employ a rigorous definition of server unavailability. When a server does not meet the performance requirement, it is deemed as unavailable by the load balancer. These types of unavailability are rarely reported by data centers and HPC systems. Unfortunately, the unavailability logs do not provide a causal breakdown of failure events.

Unavailability is mostly transient. Figure 5.2a shows a CDF of the durations of unavailabilities. We observe that unavailabilities can last between 20 minutes and 24 hours with a median duration of 200 minutes. These short unavailabilities are mostly caused by performance degradation, such as unexpected server overload and software issues (e.g., application/kernel bugs or upgrades). Besides, we observe a long tail of unavailability durations, with around 16% exceeding 24 hours and 2% exceeding an entire week. These cases may be related to hardware issues. Qualitatively, our observations are similar to storage systems in the sense that unavailabilities are common, and most unavailabilities are not permanent.

5.2.3 Mitigating unavailability is challenging

Upon detecting an unavailability, the load balancer removes the corresponding server from the consistent hash ring and reassigns their buckets to other servers [219]. We evaluate how a bucket's object miss ratio is affected by unavailability using the video trace. Figure 5.2b shows that the object miss ratio in a CDN cluster *without any redundancy* increases by more than 2× relative to no unavailability over the same time period. *This spike disproportionally affects a small group of content providers because of bucket-based routing* (subsection 5.2.1). The high latency resulting from cache misses can lead to SLA violations.

The state-of-the-art mitigation technique for server unavailability at large CDNs is *replicating* buckets across two servers[2]. When one server becomes unavailable, requests are routed to the other server, which is likely to hold the object. Figure 5.2b shows that replication reduces the intensity of the miss ratio spike. However, we find that

[2]For operational flexibility, CDNs do not replicate servers as primary/backup. CDNs implement replication using additional virtual nodes for a bucket on the consistent hash ring [178, 246].

replication does not remove the miss ratio spike. *In contrast to storage systems, where replication guarantees durability, in CDN clusters, servers perform cache evictions independently.* Objects that are admitted to two caches at the same time may be evicted at different times. This is particularly common if the two caches evict objects at very different rates, making replication ineffective. We next discuss why this case is more common than one might expect.

5.2.4 The need for write load balancing

We measure the read and write load balance across servers in a CDN cluster. To make the analysis independent from eviction decisions, we present the read and write rates based on compulsory misses from the web trace Table 5.1 shows that the server with the highest read load serves 1.3× more traffic than the server with the lowest read load. The server with the highest write load writes around 2.5× more bytes than the server with the lowest write load.

Write load imbalance causes three problems. First, imbalance reduces the effectiveness of replication. A server with a 2.5× higher write rate also has a 2.5× higher eviction rate. So, a newly admitted object will traverse the cache with the highest write load 2.5× faster than the one with the least write load. Consequently, buckets mapped to these servers will have many objects for which only a single copy is cached in the cluster. We find that for 25% of objects, only a single copy exists in the cluster, which leads to the miss ratio spike observed during unavailabilities (Figure 5.2b). Second, SSD write load imbalance often causes high tail latency. Specifically, high write rates frequently trigger garbage collection, which can delay subsequent reads by tens of milliseconds [47, 352, 353, 367]. Third, the imbalance can lead to short SSD lifetimes due to concentrated writes on some SSDs, and thus higher replacement rates [41, 320], which increases CDN cost (section 5.1).

5.3 C2DN System Design

C2DN's design goals are to: (1) eliminate miss ratio spikes caused by server unavailability and (2) balance write loads across servers in the cluster. Erasure coding is a promising tool to improve availability under server unavailability. We first describe a naive implementation, called C2DN-NoRebal, based on a straightforward application of erasure coding (subsection 5.3.1). C2DN-NoRebal fails to achieve the targeted goals, and we identify write and eviction imbalance as the key challenge. We then describe a new technique to overcome this challenge (subsection 5.3.2) that exploits the unique aspects of the use of erasure coding in the context of CDNs.

5.3.1 Erasure coding and C2DN-NoRebal

Erasure coding is widely used in production storage systems for providing *high availability* with *low resource overhead* [174, 249, 278]. Conceptually, erasure coding an object involves dividing the object into K *data chunks* and creating P parity chunks, which are mathematical functions of the data chunks. Such a scheme, called a (K, P) coding scheme, enables the system to decode the full object from any K out of the $K + P$ chunks. Thus, caching $K + P$ chunks on different servers provides tolerance to P server unavailabilities. As individual chunks are only a fraction $1/K$ of the original object's size, coding reduces space overhead compared to replicating full objects[3].

As CDNs use bucket-based routing (section 5.1), coding needs to be applied at the level of buckets rather than objects. Specifically, the K data chunks of all the objects belonging to a bucket are grouped into K distinct *data buckets* respectively. Similarly, the corresponding P parity chunks are grouped into P distinct *parity buckets*. These buckets (data and parity) are each assigned to a distinct server in the cluster. Note that while the routing happens at the level of buckets, requests are still served at the level of objects. Hence we will use the term *buckets* in the context of the assignment and *chunks* in the context of serving specific objects.

The application of erasure coding to CDNs is shown in Figure 5.3. To serve a user request, a server reads one chunk from the local cache and at least $K - 1$ chunks from other servers to reconstruct the requested object. To find the location of data and parity chunks, *C2DN-NoRebal* relies on a simple extension of bucket-based consistent hashing. The location of the first chunk is the server the bucket containing the object hashes to. Then, subsequent $K + P - 1$ chunks are read from the subsequent $K + P - 1$ servers on the consistent hash ring.

Owing to the reduced storage overhead, C2DN-NoRebal provides cost benefits by

[3]The space overhead of an (K, P) coding scheme is $\frac{K+P}{K}$. For example, for $K = 3, P = 1$, space overhead is $1.33\times$ as opposed to $2\times$ in two-replication.

reducing the average byte miss ratio when compared to replication (as seen in our experiments in section 5.5). However, C2DN-NoRebal *fails to eliminate the object miss ratio spike during unavailability* (section 5.5). Specifically, we find that coded caches are even more sensitive to write load imbalance than replication. For replication, eviction rate imbalance may cause the second (backup) copy to be evicted, which is required when a server becomes unavailable. Whereas for a coded cache, eviction rate imbalance could lead to any of the individual chunks being evicted, which leads to an effect we call *partial hits*: less than K chunks of the object are cached in the cluster, and this prohibits the reconstruction of the object. A partial hit only requires fetching the missing chunks but incurs the same round-trip-time latency as a miss and thus does not provide a speedup. Further, partial hits become even more frequent during server unavailability, thus deeming C2DN-NoRebal less effective.

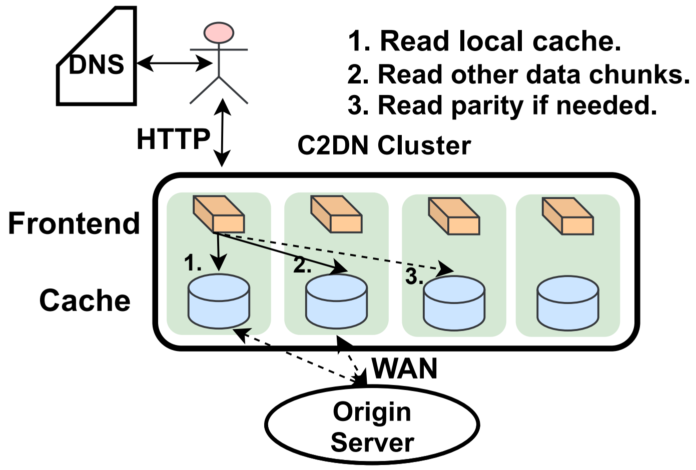


Figure 5.3: Architecture of C2DN.

5.3.2 Parity rebalance and C2DN

Having identified write imbalance as a key challenge for erasure coding in CDNs, we next show how we exploit parities in overcoming these imbalances. Our main idea is to assign *parity buckets* to servers in a way that mitigates the write load imbalance caused by *data bucket* assignment.

Like the state-of-the-art in CDNs and C2DN-NoRebal, C2DN applies consistent hashing to assign the data buckets (Figure 5.4). We define a server's data write load as the

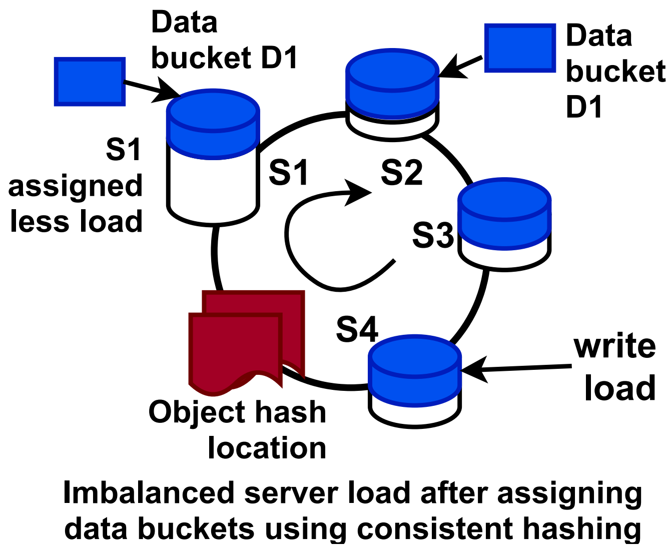


Figure 5.4: C2DN bucket assignment. C2DN assigns data buckets using consistent hashing, which guarantees a consistent mapping across unavailabilities, but causes load imbalance

number of bytes written (i.e., admitted) to the cache, counting only data buckets. We also define a bucket's parity write load as the bytes written counting only parity buckets. Every server records the data write load and each bucket's parity write load since the cluster's last unavailability event. After an unavailability event, parity buckets are reassigned by the load balancer using this information. The load balancer calculates an assignment of parity buckets to servers to balance the write load. This assignment is a non-trivial calculation as not every assignment is feasible: parity chunks cannot be assigned to a server that holds a data chunk of the same object. In general, C2DN's parity bucket assignment problem is NP-hard by reduction from the Generalized Assignment Problem [68].

C2DN's parity bucket assignment algorithm. We obtain an approximate solution in polynomial time using a MaxFlow formulation (Figure 5.5). The solution provides us with feasible server assignments for each parity bucket. We empirically observe that by assigning the parity bucket to the least loaded server among the feasible servers, the write load on each server is well balanced. The inputs to the algorithm are:

1. parity write load of bucket n (s_n),
2. data write load on server i (l_i),
3. total write load on the cluster (W),
4. current assignment of data buckets to servers,

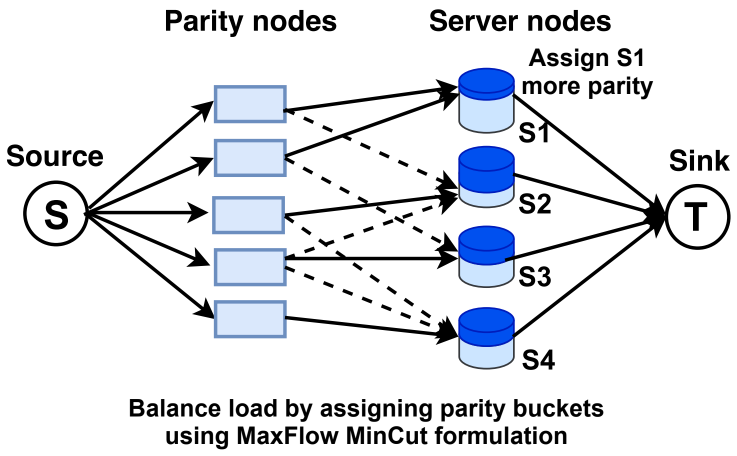

Parity nodes
Server nodes
Assign S1 more parity
Source
S
S1
S2
S3
S4
Sink
T
Balance load by assigning parity buckets using MaxFlow MinCut formulation

Figure 5.5: Parity rebalance. Write load imbalance is mitigated by assigning parity buckets to balance the load using a MaxFlow formulation.

5. available servers in the cluster ($\mathcal{A}$).

The flow graph (Figure 5.5) is constructed using a source-node (S), parity-nodes corresponding to each parity bucket, server-nodes corresponding to each server in the cluster, and a sink-node (T). We add an edge from the source-node (S) to each parity-node n with a capacity equal to the bucket's parity write load (s_n). We add edges from parity-nodes to the server-nodes if the corresponding parity bucket can be placed on that server, i.e., the data chunks of the bucket are not assigned to the server. The capacity of these edges is again the bucket's parity write load (s_n). Finally, we add edges from server-nodes to the sink-node (T) with a capacity equal to the server's remaining write load budget, which is $\max(\left\lceil \frac{W}{|\mathcal{A}|} \right\rceil - l_i, 0)$.

After solving MaxFlow(S,T), C2DN iterates over parity buckets. Each parity bucket is assigned to the least loaded server with a positive flow from the parity-node to the server-nodes. This leads to a well-balanced assignment. The assignment is also feasible as no positive flow exists between a parity bucket and the servers holding this bucket's data chunks.

Extension to heterogeneous servers. We incorporate heterogeneous servers by setting the capacity of the edge in the graph between server-nodes to sink-node (T) proportional to the size of the server.

5.3.3 C2DN resolves partial hits

Having shown how to balance write loads across servers within the cluster, we show that this is sufficient to solve the issue of partial hits. Specifically, we find that the probability of a partial hit diminishes for large caches.

We formulate our proof under the simplifying assumptions of the independent reference model (IRM[4]), which is used widely in caching analysis [9, 116]. While our proof can be extended to a range of eviction policies [223], we assume the Least-Recently-Used (LRU) policy for simplicity. We empirically observe that FIFO, which is used in open-source caches such as Apache Trafficserver [14] and our empirical evaluation in section 5.5, behaves similarly to LRU.

We remark that we *do not require explicit coordination of individual eviction decisions among the caches*. Our theorem states that under IRM, in C2DN, if one chunk of an object is present in a cache, then the other chunks are almost surely present in the other caches.

Theorem 1. *Under IRM and LRU, in C2DN, for an object with chunks $x_1, \dots, x_n$, for any $1 \leq i, j \leq n$., and as the cache size grows large*

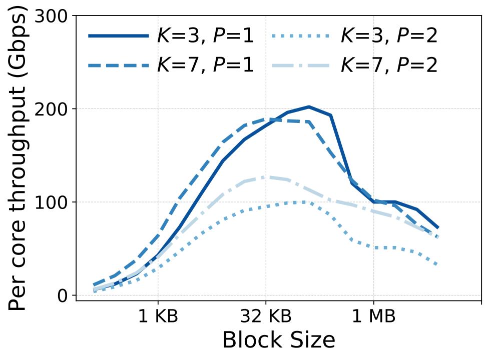
$$P[\textit{chunk } x_i \textit{ is in cache} \mid \textit{chunk } x_j \textit{ is in cache }] \to 1 \tag{5.1}$$

Our proof uses the fact that balanced write loads lead to equal *characteristic times* [116, 124, 296], which is the time it takes for a newly requested chunk to get evicted from each server's LRU list. Since data and parity chunks of an object are requested simultaneously and the characteristic time is the same, the chunks are also evicted simultaneously, and partial hits become rare. Details can be found in our original paper [373].

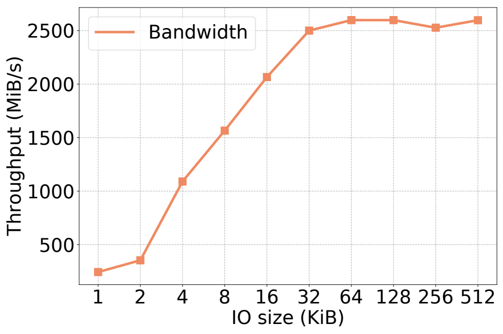
5.4 C2DN Implementation

In addition to design goals (1) and (2), C2DN's implementation seeks to (3) minimize storage/ latency/ CPU overheads and (4) remain compatible with existing systems to facilitate deployment. This entails subtle implementation challenges.

[4]In the IRM, an object i's requests arrive according to a Poisson process with a rate λ_i, independent of the other objects' requests. With recent theoretical advances [163], our proof can be extended to not assume the IRM.



(**a**) Decoding throughput



(**b**) SSD throughput

Figure 5.6: Microbenchmarks. a) With vector instruction in modern CPU, decoding is very efficient with high throughput, the sub-chunk size to achieve maximum throughput across configurations is around 32-64 KB. b) Modern SSD achieves maximum throughput with I/O size larger than 32 KB.

Enabling transparent coding. A key architectural question is which system component encodes and decodes objects into/from data and parity chunks. A natural choice might be to encode objects at the origin servers. However, this would require changes to thousands of heterogeneous origin software stacks — a barrier to deployment. Additionally, encoding at the origin would increase origin traffic as each cache miss needs to fetch both data and parity chunks, e.g., with $K = 3$, $P = 1$ the origin traffic would increase by 33%. Thus, C2DN fetches uncoded objects from origins and encodes chunks within the CDN cluster. Additionally, any decoding operation is also performed within the cluster for transparency on the client side.

Selective erasure coding. While encoding and decoding are fast due to broad CPU support for vector operations, the overhead of fetching becomes significant for small objects. As the majority of requests are for small objects (subsection 5.2.1), we can reduce processing overheads by using replication for small objects. C2DN applies coding to large objects, which account for most of the production cluster's cache space (subsection 5.2.1). Of course, with selective coding, we now need to count uncoded objects as part of the data write load in subsection 5.3.2.

To decide the size threshold of coding, we perform two microbenchmarks studying how coding block size affects coding throughput and SSD bandwidth. Figure 5.6a shows that even on a five-year-old Skylake Xeon, decoding is very efficient with per-core throughput over 200 Gbps (data fits in CPU cache) at a block size of 32 KB. This benchmark result suggests that decoding will not be a bottleneck at a reasonable block size (e.g., 32 KB) compared to NIC bandwidth. Figure 5.6b shows the relationship between SSD bandwidth and I/O size (setup as in section 5.5). We again find that a block size of 32-64 KB achieves the peak SSD bandwidth. Based on these results, C2DN codes object larger than 128KB so that each chunk is at least 42KB for a (3, 1) coding scheme.

This hybrid approach enables load balancing and space efficiency with no overhead for most requests. One might ask why C2DN relies on replication for small objects after section 5.1 showed that replication continues to suffer from miss ratio spikes. We find that erasure coding large objects is sufficient to balance eviction rates (using C2DN's parity rebalance), making replication effective for small objects.

Parity rebalance and parity look up. As described in § 5.3.2, C2DN formulates the parity bucket assignment problem as a Max Flow problem. We solve the problem using Google-OR [272], which implements the push-relabel algorithm [83]. The time complexity of this algorithm is $O(n_{node}^2 * \sqrt{n_{edge}})$. where n_{node} is the number of nodes (#buckets + #servers) and n_{edge} is the number of edges (≈#buckets × #servers). In production systems, #buckets is in the range of 100s for a 10-server cluster. Thus, the time complexity simplifies to $O(\#buckets^3)$. Empirically, we observe low run times as well, for e.g., for 100 buckets and 10 servers, C2DN's parity bucket assignment runs within 50 μs. Also, note that the parity bucket mapping is calculated in the background (off the critical path) and only when there is an unavailability event. From our analysis, we observe around 5.6 unavailability events on an average day.

Support for large file serving, HTTP streaming, and byte-range requests. To minimize latency, CDNs stream large objects to clients. We achieve compatibility with streaming by subdividing data and parity chunks (for very large objects) into smaller parts which we call sub-chunks. C2DN's encoding and decoding work on the sub-chunk level as shown in Figure 5.7. We implement streaming by serving sub-chunks as they become available. For byte-range requests, C2DN fetches the sub-chunks overlapping with the requested byte-range.

Delayed fetch of parity sub-chunks. C2DN can serve a request with any K sub-chunks (out of $K + P$). Because serving with data sub-chunks requires no decoding, C2DN first fetches all K data sub-chunks. C2DN only fetches parity sub-chunks after a heuristic wait period to overcome stragglers. We record the time until the first data sub-chunk

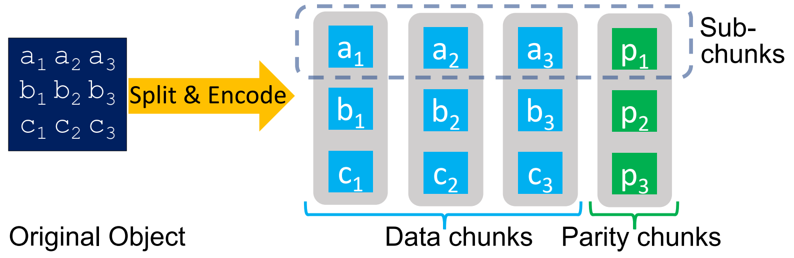


Figure 5.7: Support for HTTP streaming. C2DN efficiently supports HTTP streaming and byte-range requests by splitting large files into sub-chunks and performs coding on sub-chunks level.

is returned. If, after an additional 20% wait time, fewer than K data sub-chunks have arrived, C2DN fetches parity sub-chunks.

Hot object cache (HOC). To facilitate serving hot objects, C2DN caches decoded sub-chunks in DRAM so that if an object is popular, it will be served directly and efficiently from DRAM, thus avoiding fetching and possible decoding.

Metadata lookups. In the case of a HOC miss, C2DN needs to know if the object was encoded or replicated. Storage systems can rely on external metadata for this case, which is not available in CDNs. Thus, C2DN stores metadata with each cached object, indicating whether the object is coded or not. On a HOC miss, C2DN first looks up the object in its *local SSD cache*. If the metadata indicates a coded object, C2DN fetches chunks from other caching servers within the cluster. In the case of a local cache miss, C2DN retrieves the object from other CDN clusters or the origin servers, then C2DN serves the object to the end-user, stores it locally, and encodes or replicates based on the object size.

Table 5.2: Object and byte miss ratio from prototype

System	Replication(CDN)	C2DN	C2DN reduction
Object miss ratio	0.242	0.227	6.4%
Byte miss ratio	0.118	0.105	11%

5.5 Evaluation

To study a more comprehensive parameter range, we use simulations.

The highlights of our evaluation are: (1) C2DN eliminates miss ratio spikes after unavailabilities. Additionally, C2DN reduces byte miss ratio by 11%, enabling significant bandwidth cost savings at scale. (2) C2DN reduces write load imbalance by 99%. (3) C2DN achieves the same latency, and lower average SSD write rates with only a 14% increase in CPU utilization.

5.5.1 Experimental methodology and setup

Traces. We evaluate C2DN using the two production traces described in section 5.2. In the following sections, we focus on the video trace and the results for the web trace are similar.

Prototype evaluation setup. We emulate a CDN's geographic distribution by placing sets of clients, a 10-server CDN cluster, and an origin data center in different AWS regions. CDN servers use i3en.6xlarge VMs with 80 GB in-memory cache and 10 TB disk cache. To reduce WAN monetary bandwidth costs of the experiments, we measure latency via spatial sampling [332, 333] for 2% of requests. The remaining requests are generated in the same region.

Unless specified otherwise, we use Reed-Solomon codes ($K = 3, P = 1$). We only code objects larger than 128 KB (section 5.3). The prototype experiments use four days of requests to warm up caches. Measurements are then taken for three days of requests. This corresponds to replaying 1.18 PB of traffic in total from local and remote clients in each prototype experiment.

Simulation setup. We implement a request-level cluster simulator. While the simulator does not capture system overheads, it is useful in comparing various schemes for the full duration of the trace and for various cache sizes (which are prohibitively expensive to perform using prototype experiments.) Simulations use 18-day long traces (compared to 7 days with the prototype). Unless otherwise stated, the simulator uses the same configuration as the prototype.

Baselines. We compare C2DN to three baselines. **(1) No-replication** does not provide fault tolerance and incurs no space overhead. **(2) Replication (CDN)** replicates each object with two replicas. We use the (CDN) suffix as this is most similar to the approach deployed today. **(3) C2DN-NoRebal** a C2DN variant based on consistent hashing without parity rebalance. In addition to C2DN, which uses one parity chunk and tolerates one unavailability, we have also evaluated C2DN-n5k3 and C2DN-n6k3, which uses two and three parity chunks and can tolerate two and three unavailabilities, respectively.

5.5.2 Miss ratio without unavailability

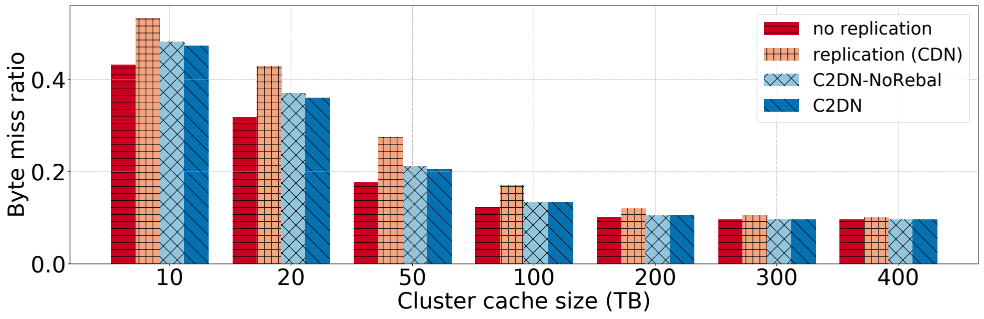


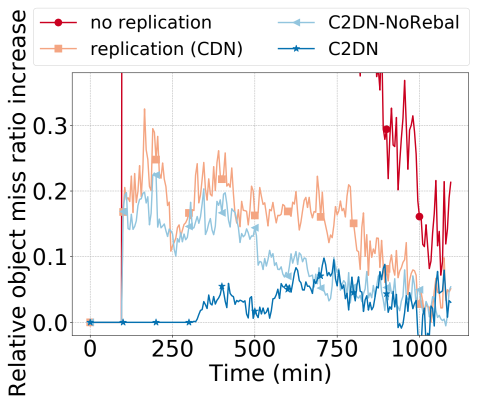
Figure 5.8: Byte miss ratio of the four systems.

We evaluate miss ratios of the competing systems under normal operation, i.e., *without unavailability*. Table 5.2 shows the object miss ratio and byte miss ratio of Replication (CDN) and C2DN obtained from the prototype experiments. We observe that C2DN reduces object miss ratio by 6.4% and byte miss ratio by 11.0%. These improvements are direct results of the reduced storage overhead in C2DN. On a large scale, these improvements lead to significant bandwidth savings.

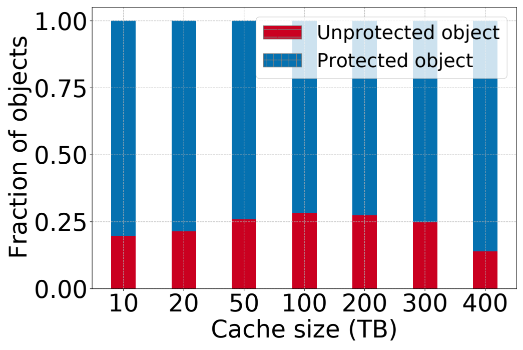
To understand the sensitivity of byte miss ratio improvements to cache size, we show simulation results in Figure 5.8. For smaller cache sizes, C2DN improves byte miss ratios by up to 20%. Benefits diminish for cache sizes above 200 TB (5× production cache size). For object miss ratios, the effect is qualitatively similar. Overall, the reduction in the miss ratio bridges the efficiency gap between No-replication and Replication (CDN) and reduces the overhead of providing redundancy in CDN edge clusters.

We also observe that C2DN improves miss ratios compared to C2DN-NoRebal because C2DN balances the write loads (eviction rates) across servers and reduces the probability of partial hits. However, this effect is small, suggesting that most of C2DN's miss ratio reduction comes from reduced storage overhead. The advantage of C2DN over C2DN-NoRebal will become clear in the following section, where we find that C2DN-NoRebal does not provide effective fault tolerance.

5.5.3 Miss ratio under unavailability



(**a**) Replicated CDN mitigates unavailability, but still has a spike after unavailability. C2DN mitigates the unavailability spike.



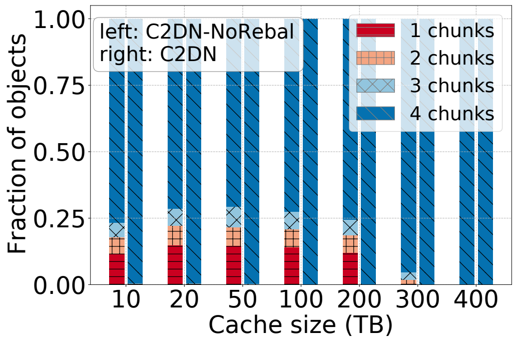
(**b**) Servers in the Replicated CDN evict objects independently, and due to write imbalance this leads to unprotected objects.

Figure 5.9: System performance under unavailability.

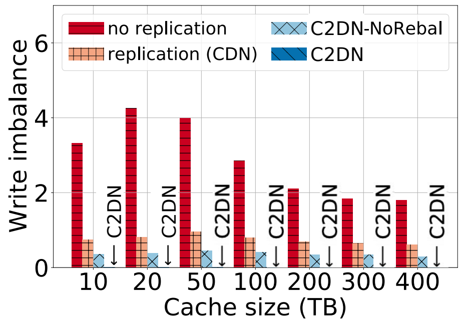
We now consider unavailabilities and evaluate the object miss ratio as the primary performance metric affecting latency and speedup. The first experiment introduces single unavailability after warming up the cache. We then measure the relative object miss ratio change: $\frac{mr(un)-mr(av)}{mr(av)}$ for each 5-minute time interval, where $mr(un)$ and $mr(av)$ stand for miss ratio with unavailability and without unavailability, respectively. A second experiment considers two simultaneous unavailabilities.

Figure 5.9a show the relative object miss ratio increase where the single unavailability event occurs 100 minutes after warmup. As expected, No-replication does not provide fault tolerance, leading to a large (2.2× as seen in 5.2b) miss ratio spike. Replication (CDN) and C2DN-NoRebal have similar performance with 25% miss ratio spikes.

The miss ratio of C2DN is not affected for several hours after the unavailability event. This is because C2DN with one parity chunk can tolerate one unavailability effectively. In the long term, miss ratios for all systems increase as the cluster's total capacity is reduced. For C2DN, the increase in the miss ratio becomes visible only after around 300 minutes past unavailability. During unavailability, data that should be written to the unavailable servers are written to the other available servers. The extra writes take a long time to impact the miss ratio of a cluster with a large cache size. We remark that the exact length of such no performance degradation is not fixed and is dependent on the trace.



(**a**) Naive coding has similar problems as replication; C2DN solves this problem by parity rebalance.



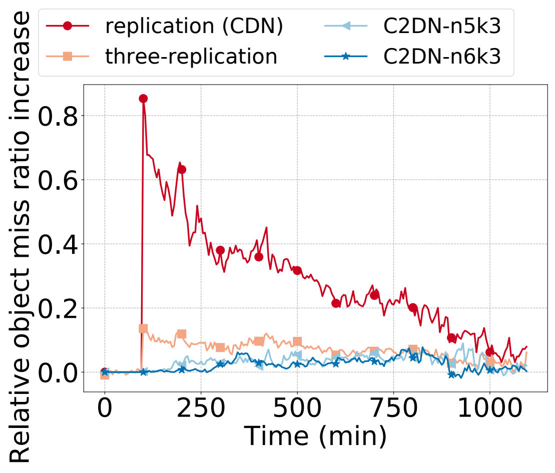
(**b**) Write load imbalance for different systems across various cache sizes.

Figure 5.10: System performance under unavailability (continued).

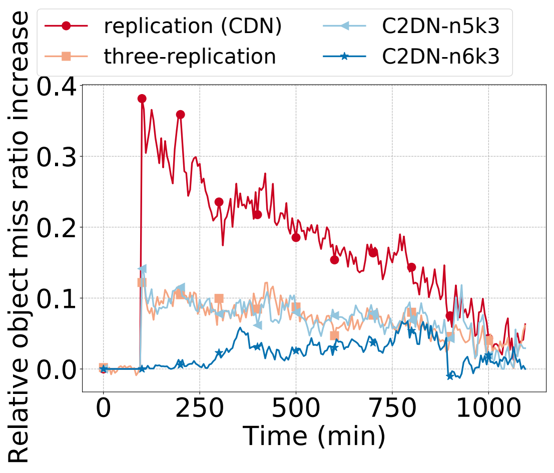
The reason for the miss ratio spike in Replication (CDN) and C2DN-NoRebal— despite using redundancy — is the severe write and eviction rate imbalance in these systems (subsection 5.2.3 and 5.2.4). This imbalance leads to *unprotected* objects: an object is unprotected if only a single copy is cached in the cluster. For C2DN-NoRebal, unprotected objects are objects with fewer than $K + 1$ chunks cached in the cluster.

Figure 5.9b shows the fraction of (un)protected objects in Replication (CDN). We observe that more than 25% of objects can be unprotected. The fraction of unprotected objects initially increases with cache size and then decreases. This pattern is because only highly popular objects are cached when the cache size is small, and hence the chance of having both replicas is higher. On the other hand, replicas are less likely to be evicted when the cache size is very large. Figure 5.10a shows the fraction of unprotected objects in C2DN-NoRebal and C2DN. Since K =3 and P =1, objects with fewer than 4 chunks are unprotected. For caches smaller than 300 TB, up to 24% of the objects in C2DN-NoRebal are unprotected. In contrast, C2DN protects nearly 100% of cached objects across all cache sizes and effectively eliminates miss ratio spikes.

So far, we have only focused on one unavailability. When a CDN operator seeks to tolerate more than one unavailability, C2DN's advantage over replication increases as the space requirements for erasure coding scale significantly better. As an empirical data point, we consider two unavailabilities and compare two-replication and three-replication with C2DN-n5k3 and C2DN-n6k3. C2DN-n5k3 (C2DN-n6k3) uses two (three) parity chunks with 66% (100%) storage overhead and can tolerate two (three) unavailabilities.



(**a**) Affected server 1



(**b**) Affected server 2

Figure 5.11: With two simultaneous unavailabilities, two replication (CDN) shows a big spike when the unavailability happens. Three replication and C2DN-n5k3 still show a small spike due to evicted replica/chunk. C2DN-n6k3 completely eliminates the spike.

In contrast, two-replication and three-replication tolerate one and two unavailabilities with 100% and 200% storage overhead, respectively.

Figure 5.11 shows that compared to two-replication, C2DN-n5k3 and three-replication significantly reduce the miss ratio spike from over 80% to less than 20%. Furthermore, the miss ratio spike disappears entirely with C2DN-n6k3, which has the same storage overhead as two-replication.

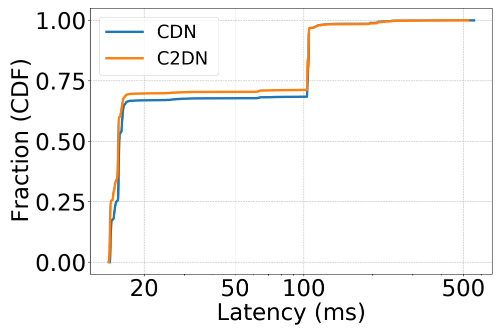
5.5.4 Write (Read) load balancing

Table 5.3: Write load on servers in Replication (CDN) and C2DN

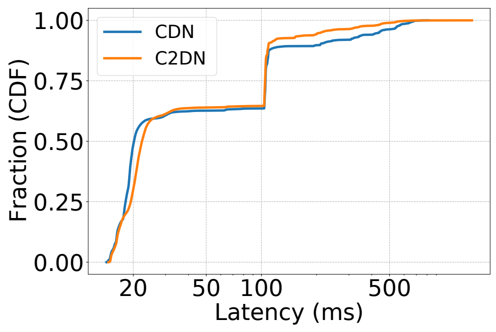
System/server load	Max	Min	Mean	Max/min
CDN write (TB)	16.83	9.26	13.48	1.82
C2DN write (TB)	8.44	8.40	8.42	1.00

We quantify how well systems balance write load across servers. Balancing writes is the key to mitigating miss ratio spikes and helps control SSD tail latency and endurance.

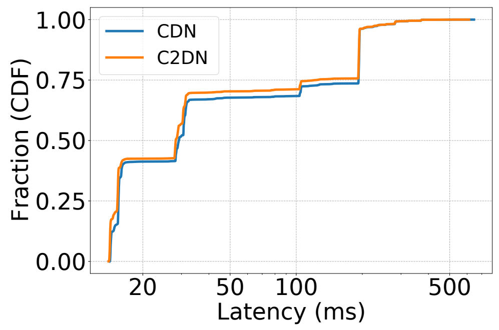
Table 5.3 shows bytes written per server in our prototype experiments. The busiest



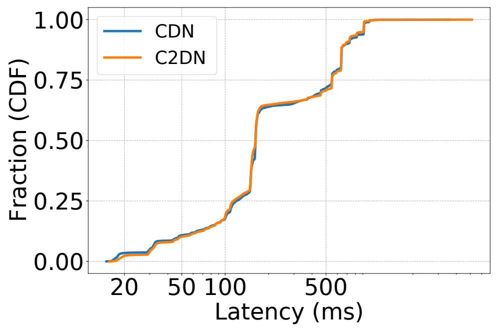
(**a**) First byte small objects



(**b**) First byte large objects



(**c**) Full response small objects



(**d**) Full response large objects

Figure 5.12: First-byte and full response latency of serving small and large objects in CDN and C2DN.

server in Replication (CDN) writes 16.8 TB compared to 8.4 TB for the busiest server in C2DN. With half the write rate, C2DN may double SSD lifetime and reduce tail latency by up to an order of magnitude [366]. The write imbalance in Replication (CDN) between peak and minimum write rate is 1.82×. In contrast, the write imbalance in C2DN is less than 1.005×. We also observe that C2DN reduces read imbalance from 1.69× for Replication (CDN) to 1.34×. The read imbalance in C2DN remains as parity rebalancing (subsection 5.3.2) focuses exclusively on write rate.

We further explore the effects of load balancing across various cache sizes using simulations. If M is the write (read) load on the server with maximum write (read) load and m is the minimum write (read) load across the servers, then write (read) load imbalance $= \frac{M-m}{m}$. Figure 5.10b shows that C2DN eliminates write imbalance for all cache sizes. When averaged across cache sizes, C2DN reduces the **write** load imbalance by 99.9% compared to No-replication, 99.8% compared to Replication (CDN), and 99.5% compared to C2DN-NoRebal. C2DN also reduces the **read** load imbalance: by 93.9%

compared to No-replication, 78.9% compared to Replication (CDN), and 70.5% compared to C2DN-NoRebal on an average across the different cache sizes.

5.5.5 Latency

We quantify potential latency overheads by measuring the time-to-first-byte (TTFB) and content download time (CDT) of our prototype implementations of C2DN and Replication (CDN). In each case, we separately measure the latency distribution for objects below the 128KB coding threshold ("small" objects) and for objects above the threshold ("large" objects). Figure 5.12a and Figure 5.12b show the cumulative distributions of TTFB for small and large objects, respectively. For small objects, we find that the TTFB distributions for C2DN and Replication (CDN) are similar, as expected: C2DN does not code these objects. C2DN slightly improves the TTFB distribution (shifting to the left) due to its lower object miss ratio. For large objects, we find about a 1 ms overhead in TTFB at low percentiles (25th-60th percentile). The slight increase is for cache hits due to fetching the first sub-chunk from K servers before serving the object.

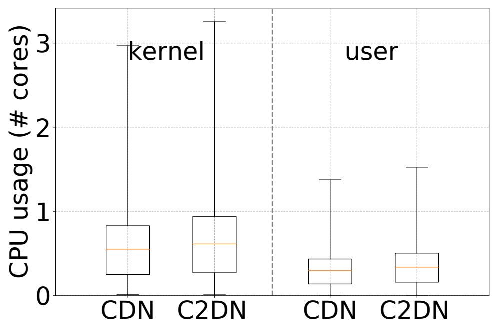
We now consider CDT. In practice, this metric is more relevant for large objects than the TTFB. Figure 5.12c and Figure 5.12d show a cumulative distribution of the content download time for small and large objects, respectively. Again we find that small objects behave similarly in Replication (CDN) and C2DN, with slightly better latency for C2DN due to a lower object miss ratio. For large objects, C2DN and Replication (CDN) have a similar CDT. The overheads of fetching chunks are hidden by our streaming implementation based on sub-chunks (section 5.4).

We remark that C2DN improves the tail latency in all cases (barely visible in the CDFs). For example, C2DN reduces the P90 TTFB by up to 3× compared to Replication (CDN). We attribute this to a lower miss ratio and the mitigation of stragglers using parity chunks to serve requests. This is as expected based on prior work on using coding to reduce tail latency [281].

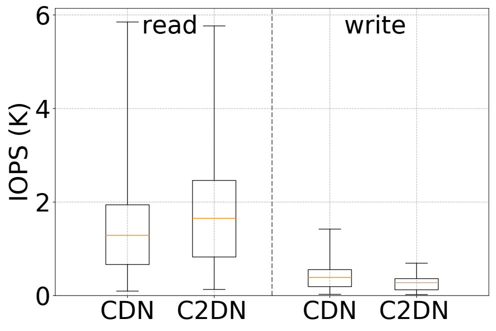
5.5.6 Overhead assessment

We quantify the resource overheads of our C2DN prototype.

CPU usage. Figure 5.13a measures CPU utilization in fractional CPU cores for userspace



(a) CPU usage



(b) Disk IOPS

Figure 5.13: Resource usage. C2DN uses slightly more CPU resources and slightly more read disk IOPS than CDN, however, C2DN reduces write disk IOPS, especially at peak.

and kernel tasks, respectively. C2DN generally leads to higher CPU usage. The userspace CPU usage is higher due to the encoding and decoding of objects, and the kernel CPU usage is higher due to additional network and disk I/Os. Overall, CPU usage increases by 14% on average with a similar increase in the kernel and userspace CPU usage.

The increase in the CPU overhead is small as C2DN performs the encoding and decoding operation only on a fraction of requests. For the current coding size threshold of 128 KB, the number of requests served with coded objects is around 50%, while the number of bytes served using coded objects is close to 90%. Also, recall that most requests for coded objects do not need to be decoded as the object is recreated by concatenating data chunks in the output buffer. Decoding only happens in the case of stragglers and partial hits. In fact, only 6% of requests require decoding in our experiments. These cases happen primarily due to the straggler problem (individual slow servers); actual data chunk misses (partial hits) occur for less than 0.6% of requests. A future version of C2DN may further reduce CPU overheads by using kernel-bypass networking or increasing the object size threshold for coding. Increasing the threshold can happen with minimal side effects, as we show in the next section.

Disk usage. Figure 5.13b compares disk IOPS of Replication (CDN) and C2DN for reads and writes. For reads, we observe that C2DN uses 23% more IOPS in the mean and less at the tail. Read-IOPS increase by 2% at the P99 and decrease by 11% at the P99.9 (we calculate this percentile across time and servers). For writes, C2DN uses 24% fewer write IOPS in the mean. The tail write IOPS decreases by 46% at the P99 and 50% at the P99.9. The read IOPS increases because C2DN fetches at least $K = 3$ chunks to serve an object if coded. However, due to 1) most of the requests being for small uncoded objects, and 2) the presence of DRAM hot object cache, the increase in reading IOPS is much smaller than $3\times$. While the mean read IOPS increases, the peak read IOPS is similar or lower in C2DN. We attribute this to better load balancing in C2DN. Write IOPS in C2DN is significantly reduced when compared to Replication (CDN). C2DN has a lower storage overhead than Replication (CDN) and thus writes less to disk. In addition, the improvement in the miss ratio that C2DN provides further reduces the number of write operations. Besides, C2DN also improves the tail write IOPS, which is due to a better load balancing strategy of erasure coding and parity rebalance.

Intra-cluster network usage. C2DN uses network bandwidth within the cluster, about 0.9 Gbps in the mean and 2.3 Gbps at the P95. In conversations with CDN operators, this internal bandwidth usage is feasible for production clusters, as these links generally show little usage. For example, production CDN clusters use dedicated 10-Gbps-NICs for communication within the cluster.

5.5.7 Sensitivity analysis

We discuss the sensitivity of C2DN to its parameters.

Coding size threshold. The size threshold for coding impacts the performance in multiple ways. By reducing the size threshold, C2DN encodes more objects, improving cache space usage and load balance across cluster servers. At the same time, it leads to more CPU and I/Os (due to coding and fetching) and increases the latency for small objects. The size distribution in Figure 5.1a shows that small objects contribute a small fraction of cache space usage. Thus, the potential benefit of coding diminishes as we decrease the size threshold for coding. At the same time, C2DN would use more cluster resources. We observe that reducing the size threshold to below 128 KB does not significantly benefit the object and byte miss ratio. Increasing the size threshold to over 8 MB increases the byte miss ratio by 2.79% and the write load imbalance by 258%. We believe that 128 KB is a good tradeoff for our production traces.

Coding parameter K. Most of this section assumed C2DN configured with $K = 3$. We explore the impact of parameters K and P on miss ratio and write load balancing. We find that increasing K and keeping P constant reduces miss ratios for C2DN but increases miss ratios for C2DN-NoRebal. When adding chunks, the probability of getting partial hits increases for C2DN-NoRebal due to unbalanced eviction rates between servers. Because C2DN uses parity rebalance to achieve similar eviction rates between servers, the miss ratio decreases with increasing K due to lower storage overhead. While the impact of coding parameters has different impacts on miss ratios for C2DN-NoRebal and C2DN, the impact on load balancing is similar, as K increases, because an object is broken into more (and smaller) chunks, both the read and write load imbalance in C2DN-NoRebal and C2DN reduce.

Different workloads. Throughout this section, we have used the video trace. We repeated our evaluation for the week-long web trace (section 5.2). Compared to the video trace, the web trace has a significantly smaller working set. The video trace has a compulsory byte miss ratio of 0.1 and a compulsory object miss ratio of 0.21. In the web trace, the compulsory miss ratio is 0.06 for both byte and object miss ratios. In addition, compared to the video trace, the web trace has a more diverse object size range, as shown in Figure 5.1a. Less than 10% of large objects (possibly large software) contribute to more than 90% of the cache space usage. Therefore, the fraction of requests that require coding is significantly smaller. In prototype experiments with the web trace, only 3% of all requests are served coded. However, the 3% of requests account for 80% of served traffic. As a comparison, in the video trace, the prototype serves about 50% of requests from coded objects (with only 6% requiring decoding). Consequently, coding overheads on the web trace are negligible. In terms of the miss ratio, we observe a 10% reduction in object miss ratio and a 6% reduction in byte miss ratio. The write imbalance for Replication (CDN) is 1.72×, which is reduced to 1.03× in C2DN. The read imbalance for Replication (CDN) is 4.8×, which is reduced to 2.5× in C2DN.

Different eviction algorithms. Throughout this section, we have used FIFO as the eviction algorithm for the cache. FIFO provides stable performance on SSDs and extends the lifetime of an SSD by minimizing device write amplification [41, 113, 320]. Many open source caches such as Apache Trafficserver [14] and Varnish [328] use FIFO. To understand the impact of the eviction algorithm, we evaluate the Least-recently-used algorithm (LRU) using simulation. We observe a slight reduction in both object and byte miss ratios for all systems. All other results are qualitatively and quantitatively the same.

Variants of replication. Besides two-replication for all objects, CDNs have explored systems that replicate based on popularity. Specifically, only popular objects are replicated on two servers to reduce space overheads. As might be expected from our findings that write imbalance matters, popularity-based replication does not provide good fault tolerance. In simulation experiments, we observe object miss ratio spikes by 82%. Interestingly, we also observe that popularity-based replication leads to an even worse load imbalance than Replication (CDN), which explains the high miss ratio spike.

5.6 Discussion

DNS vs anycast-based CDN request routing. Different CDNs use different global load-balancing architectures. Akamai is well known for its DNS architecture [300]. Limelight, Wikipedia [347], and Cloudflare rely on anycast. While these designs have different performance implications, both rely on algorithms like consistent hashing. In DNS-based systems, consistent hashing is applied by the cluster-local load balancer to return the IP of the server responsible for the shard. Anycast-based systems typically route requests to any server in a cluster, and the server uses consistent hashing to identify another server that likely stores the object. Server unavailability, storage overheads of redundancy, and write imbalance are important problems in all CDN designs. While the cluster-local load balancer in our prototype relies on DNS, the principle design components of C2DN can be equally applied in anycast systems. We also expect that C2DN's benefits will transfer with similar quantitative improvements.

Larger clusters and multiple unavailabilities. In clusters of large size, multiple concurrent unavailabilities are not uncommon. As evaluated in section 5.5, we find that C2DN is more effective in this setting as erasure coding is more efficient at tolerating multiple unavailabilities than replication. For large clusters, server unavailabilities become more common. We thus recommend either using a coding scheme with more parity chunks or handling the cluster as multiple smaller clusters.

5.7 Related work

While there is extensive work on caching, coding, load balancing, and flash caching, our work is uniquely positioned at the intersection of these areas. We discuss work by area.

Erasure coding in storage systems. Prior work has characterized the cost advantage offered by coding over replication in achieving data durability in distributed storage systems [342, 391]. Erasure codes are deployed in RAID [270], network-attached-storage [8], peer-to-peer storage systems [186, 197, 286, 359], in-memory key-value store [79, 380, 382], and distributed storage systems [249, 277, 279, 343]. Coding for CDNs differs due to the unique interplay of coding and caching and the two-sided transparency requirement (section 5.3). Additionally, CDNs employ coding for different reasons (performance) than storage systems (durability), which magnifies overhead concerns.

Caching for coded file systems. Several recent works have explored augmenting erasure-coded storage systems with a cache to reduce latency [6, 141, 210, 281]. Aggarwal et al. [6] proposed augmenting erasure-coded disk-based storage systems with an in-memory cache at the proxy or the client-side that cache encoded chunks. Halalai et al.[141] propose augmenting geo-distributed erasure-coded storage systems by caching a fraction of the coded chunks in different geo-locations to alleviate the latency impact of fetching chunks from remote geo-locations. EC-Cache [281] employs erasure coding in the in-memory layer of a tiered distributed file system such as Alluxio (formerly [195]). Although EC-Cache is technically a cache, there is no interaction between coding and caching in EC-Cache since it operates in scenarios where the entire working set fits in memory, i.e., no evictions are considered. In contrast, C2DN focuses on CDN clusters with working sets in the hundreds of TB and starkly different tradeoffs, workload characteristics, and challenges as compared to file systems. In the area of cooperative caching [13, 149, 298], nodes synchronize caching decisions via explicit communication. In contrast, C2DN proves that explicit communication is not required to synchronize the eviction of the K chunks, which significantly decreases overheads.

Chunking and caching. Prior work has explored the challenge of serving large files over HTTP, e.g., CoDeeN [339]. Similar to C2DN, CoDeeN breaks a large file into smaller chunks. A chunk cache miss does not require transferring the whole large file from the origin. In contrast to CodeeN, C2DN addresses unavailability tolerance, which is not provided by chunking alone.

Load balancing. Load balancing and sharding are well-studied topics [4, 5, 58, 59, 112, 126, 139]. To reduce the load imbalance, John et al. study the power of two choices that reduces the imbalance [58]. In addition, to serve skewed workloads, Fan et al. [117] study the effect of using a small and fast popularity-based cache to reduce load imbalance between different caches in a large backend pool. Yu-ju et al. [150] designed SPORE to use a self-adapting, popularity-based replication to mitigate load imbalance. Rashmi et al. [281]used erasure coding to reduce read load imbalance for large object in-memory cache. In summary, prior work on load balancing focuses on *read* load balancing, with little attention paid to *write* load balancing.

Load imbalance in consistent hashing can be solved with additional lookups via probing [245]. Unfortunately, these lookups are costly in CDNs (particularly for DNS-based systems). Additionally, this approach cannot be applied for erasure-coded caches due to the constraint that parity chunks should not be colocated with data chunks. In contrast to using load balancing to achieve a similar SSD replacement time, Mahesh et al. [30] used parity placement to achieve differential SSD ages so that SSDs of a disk array fail at different times.

Flash cache endurance. Flash caching is an active and challenging research area. A line of work [190, 225, 262, 299, 304, 305, 320] shows how eviction policies can be efficiently implemented on flash. Flashield [113] proposes to extend SSD lifetime via smart admission policies. All these systems focus on a single SSD. Our work focuses on wear-leveling across servers in a cluster, which significantly extends the lifetime of a cluster.

5.8 Chapter Summary

Content Delivery Networks (CDNs) carry more than 70% of Internet traffic and continue to grow [88]. Large CDNs achieve this by operating thousands of clusters deployed worldwide, allowing users to download content with low network latency.

In this chapter, we present C2DN, a CDN design that achieves both high availability and high resource efficiency. To achieve high resource efficiency, we apply erasure coding to large cached objects. This requires overcoming multiple CDN-specific challenges, such as the eviction of object chunks due to write rate imbalances. In fact, we show that a naive application of erasure coding fails to achieve the goal. The core of our design is a new technique that enables CDNs to balance eviction rates and write loads across servers in each cluster. We exploit the fact that erasure coding enables more flexibility in assigning chunks to multiple servers. Our key insight here is that the chunk assignment can be reduced to a known mathematical optimization problem called Max Flow Problem.

The core contributions of this chapter are a novel chunk placement scheme for consistent-hashing-based load balancing in CDN clusters and a low-overhead implementation of erasure coding for CDNs that can serve the different traffic requirements of production systems. Specifically, by solving an instance of the Max Flow problem, we assign objects with *near-optimal balance* in eviction and write rates for CDN servers and their SSDs. As a consequence, C2DN can reduce storage overheads and bandwidth costs. Finally, equal write rates across servers essentially function as a cluster-wide distributed wear-leveling for the servers' SSDs, significantly extending lifetimes.

This chapter makes the following contributions.

- We show that server unavailability is common in CDN clusters by analyzing a month-long trace from over 2000 load balancers of a large CDN. We show that the state-of-the-art approach of replicating objects within a cluster does not eliminate miss ratio spikes after server unavailability events.
- We design C2DN with a hybrid redundancy scheme using replication and erasure coding, along with a novel approach for parity placement. C2DN reduces the storage overhead of providing fault tolerance and hence lowers the miss ratio. Moreover, by leveraging the parity assignment, C2DN balances the write loads and eviction rates across cache servers.
- We implement C2DN on top of the Apache Traffic Server (ATS) [14] and evaluate it using production traces. We show that C2DN provides an 11% miss ratio reduction compared to the state-of-the-art, and C2DN eliminates the miss ratio spikes caused by server unavailabilities. Further, C2DN decreases the write load imbalance between servers by 99%.

Part III

Efficient and Scalable Cache Eviction

Chapter 6

Group-level Learned Cache

The eviction algorithm is the most important component of a cache. A good cache eviction algorithm allows the cache to store more useful objects in the cache and achieve a lower miss ratio. However, determining which objects should be kept in the cache is non-trivial. Many previous works have looked into using machine learning to learn the data access patterns and compare the usefulness of objects. However, existing solutions often incur huge overheads because they have to (1) store a huge number of features together with each object and (2) run a machine learning inference on many sampled objects at *each eviction*.

This chapter introduces a new way to perform learning in caching: group-level learning, which learns at a coarser granularity so that the overheads can be amortized. Meanwhile, each object group receives more requests and, thus, more information compared to each object, which eases the learning.

6.1 Background and motivation

6.1.1 Learning in caching

To make cache eviction algorithms adaptive across workloads, cache size, and over time, recent works have explored the idea of using machine learning in caching [38, 42, 113, 291, 312, 329, 369]. These approaches can be broadly classified into three classes, which come with their pros and cons, as discussed below and summarized in Table 6.1.

Table 6.1: Comparison of different learned caches (numbers describe the example systems).

Learning approach	Example system	Learning granularity	No. Features	Overhead (Bytes/obj)	Potential efficiency	Throughput relative to FIFO
Object-level learning	LRB	object	44	189	high	0.001-0.01
Learning from simple experts	Cacheus	expert	2	32	low	0.2-0.25
Learning from distribution	LHD	workload	2	24	medium	0.2-0.25
Group-level learning	GL-Cache	object group	7	<1	high	0.3-0.8

Object-level learning

Object-level learning performs learning on each object. Multiple works have studied the prediction of object reuse distance [42, 46, 120, 208, 306, 312, 356, 364] and popularity [71, 119, 251, 390]. By predicting reuse distance, a learned cache can mimic Belady's algorithm [39], which evicts the object requested the furthest in the future using an oracle. However, predicting reuse distance is challenging [312] because an object's reuse distance is not only inherent to the object but is also affected by the access patterns of the workload. For example, the reuse distance will increase if a request burst or scan happens between the two requests to the same object. Moreover, cache workloads often follow Zipf distributions [25, 41, 66, 370]. Thus, most objects only get a limited number of requests. This leads to *limited object-level information for learning*. Meanwhile, it is these less popular objects that often affect cache efficiency [369]. As a result, existing works introduce approximations and proxies for learning reuse distance. For example, LRB [312] introduces Belady Boundary to reduce the range of reuse distance. While learning reuse distance is challenging, with careful feature engineering, large enough data, and a complex model, object-level learning may have the potential to achieve the highest hit ratio among all learned caches. However, object-level learning incurs prohibitively high storage and computation overheads.

Storage overhead. Both training and inference require extra storage. While the storage overhead of training data is often negligible with optimizations such as sampling and offloading to cheaper storage, inference data pose a significantly higher storage overhead. To make predictions on the object level, the cache needs to track features for each object. For example, LRB [312] stores 44 features (189 bytes) per object. Moreover, this large per-object metadata overhead is prohibitively high because it needs to reside in DRAM for frequent updates. Using fewer features is possible, but it leads to worse performance (section 6.3).

Computation overhead. Both training and inference add computation overhead. While training data collection and frequent re-trainings consume CPU cycles, the inference is the major source of computation overhead. The prediction in object-level learning uses dynamic features (e.g., object age), and the prediction results cannot be reused over time. Therefore, object-level learning needs to sample objects and perform inference at each write (eviction). For example, LRB samples 32 objects and copies their features to a matrix for inference for *each* eviction. In our measurement, each eviction (including feature copy, inference, and ranking) takes 200 μs on one CPU core, indicating that the cache can evict at most 5,000 objects on a single core per second. As a comparison, a production server achieves over 100,000 requests per second [259].

Learning-from-simple-experts

Several works use reinforcement learning to choose between multiple simple experts (eviction algorithms). For example, LeCaR [329] uses two experts (LRU and LFU). At each eviction, LeCaR chooses one expert to make an eviction decision based on the experts' weights. Similar designs can be found in ACME [18], FRD [269], and Cacheus[291], which use different experts and weight adjustment methods.

By using more than one algorithm for eviction, learning-from-simple-experts can adapt to changing access patterns. The overhead and efficiency of learning-from-simple-experts depend on the experts. Existing systems use simple experts and thus incur lower overhead than object-level learning. However, existing systems suffer from two problems. First, a delay exists between a bad eviction and an update on the expert's weight. The cache only discovers a bad prior eviction when the evicted object is requested again. This challenge, commonly known as "delayed rewards" in reinforcement learning [19, 142, 175, 319], limits the efficiency of caches that use learning-from-simple-experts. Second, the cache efficiency is bounded by the experts selected; an efficient policy requires a good understanding of the workload. Learning-from-simple-experts cannot leverage features that the experts do not consider. If a feature is important to the workload and not considered by any of the experts, then learning-from-simple-experts will not provide a high hit ratio. Some works used more experts [136] to capture more features. However, using more experts incurs higher overheads because it needs more computation and space to evaluate expert performance and update experts' weights.

Learning-from-distribution

The third type of learned cache models the request probability distribution and makes decisions based on the distribution. For example, LHD [38] uses the request probability distribution to calculate *hit density* (hits-per-space-consumed) as a metric for eviction. Specifically, LHD learns the request probability as a function of ages and then modulates it with size to arrive at hit density. LHD is simple yet effective and does not require expensive inference computation to compare objects. However, LHD's hit density is calculated based only on two features: age and size, and it is non-trivial to track probability with more features. Besides, LHD cannot change relative feature importance (how features are composed). Furthermore, because hit density does not change monotonically over time, LHD must sample objects to rank at each eviction, limiting its throughput due to slow random memory access.

Takeaways. We summarize the potential efficiency and overhead of the three types of learned caches in Table 6.1. We observe that object-level learning has a high potential to achieve high efficiency, but it incurs huge storage and computation overheads. Learning-from-distribution only considers a limited number of features and has lower overhead with lower potential for high efficiency. Although having a lower learning overhead, learning-from-distribution requires random sampling during *each* eviction, which limits its throughput. Learning-from-simple-experts highly depends on the experts used. Existing systems such as LeCaR and Cacheus achieve a higher hit ratio than a single expert but still leave a large hit ratio gap compared to other learned caches (subsection 6.3.3).

6.2 GL-Cache: Group-level learned cache

To enable a better trade-off between learning granularity and learning overhead, we propose learning at the level of object groups (which we term "group-level learning"). The key idea behind group-level learning is to learn the usefulness of groups of objects (called "utility"). Based on this idea, we designed **G**roup-level **L**earned **C**ache (GL-

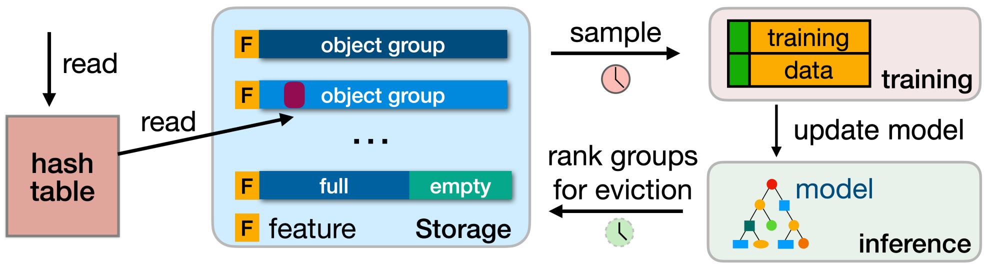


Figure 6.1: Overview of GL-Cache. Objects are clustered into groups for learning: feature tracking, model training, and inference are performed on the group level.

Cache), which learns the object-group utility and evicts the least useful object groups. We first give a high-level overview of GL-Cache's design and then go into the details of each component.

6.2.1 Overview of GL-Cache

Figure 6.1 shows an overview of GL-Cache. In GL-Cache, objects are clustered into fixed-size groups when writing to cache (subsection 6.2.3). The training module in GL-Cache collects training data online and periodically trains a model to learn the utility of object groups (subsection 6.2.5). The inference module predicts object-group utility and ranks object groups for eviction. Group-level learning requires group-level eviction: when the cache is full, object groups are evicted using a merge-based eviction which merges multiple groups into one, evicts most objects, and retains a small portion of popular objects (subsection 6.2.6).

6.2.2 Group-level learning

group-level learning has several advantages over existing learned caches:

Grouping amortizes overheads. Learning in caching incurs both computation and storage overheads. In group-level learning, these overheads are amortized over multiple objects in the group. In terms of storage, instead of adding huge per-object metadata, the metadata overhead is only added for each group. As a result, each object only incurs a tiny overhead on average (less than one byte in our implementation). The cost of inference computation is also amortized over objects. Compared to object-level learning, which performs one inference per eviction, each inference in group-level learning is used to

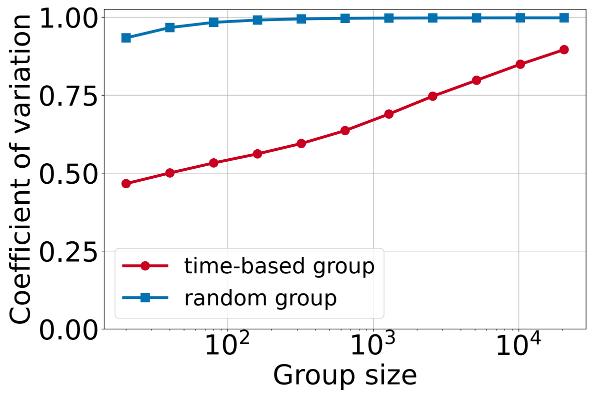


Figure 6.2: Objects grouped using write time have more similar (smaller coefficient of variation) mean reuse time than objects grouped randomly. As group size increases, write-time-based grouping becomes closer to random grouping.

evict a group of objects.

Grouping accumulates more signal. Many cache workloads follow a Zipf distribution [53, 370], and most of the objects receive very few requests. Because an object group has many objects, it often receives more requests than an individual object. More requests lead to more information on the group level compared to the object level, which makes it easier to learn and predict.

While group-level learning is promising, several challenges need to be addressed to leverage the power of learning:

- How to cluster objects into groups (subsection 6.2.3)?
- How to compare the usefulness of object groups (subsection 6.2.4)?
- How to learn the utility of object groups (subsection 6.2.5)?
- How to perform evictions at group level (subsection 6.2.6)?

While the ideas of grouping [372] and learning [312] have been studied independently in the context of caching, the combination of the two ideas in group-level learning leads to the unique challenges of *understanding, defining, and learning group utility*. We discuss these challenges and how GL-Cache overcomes them in this section.

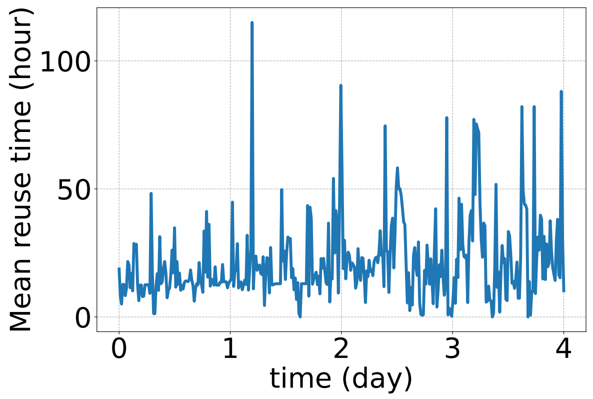


Figure 6.3: Different object groups written at different times exhibit a large variation in mean reuse time.

6.2.3 Object groups

Using group-level learning, both learning and eviction are performed at the granularity of an object group, which usually contains tens to thousands of objects. Object grouping happens when an object enters the cache, and an object should not switch groups for two reasons. First, changing groups invalidates the learning pipeline. When an object is added to or removed from a group, the accumulated group information becomes stale and cannot be used for learning. Second, in implementation, changing groups often requires copying data on the storage device. Therefore, the grouping of an object is decided when entering the cache using simple static object features. Depending on workload types, such features include time, tenant id, content type, object size, etc. In this work, we focus on grouping based on write time, which is available in all systems and hence more generalizable.

Similar to observations made in prior works [291, 372], we observe that objects written at a similar time exhibit similar behaviors. Using traces from the evaluation, we measure the mean reuse time variation of objects in (1) write-time-based groups and (2) random groups. Figure 6.2 plots the mean coefficient of variation (standard deviation over mean) of 100,000 groups for the two grouping methods at different group sizes. Compared to random groups, write-time-based groups aggregate objects with closer mean reuse time. Besides reuse time, we have similar observations on the frequency and the group utility defined below (not shown due to the space limit).

While objects *within* each write-time-based group have similar reuse, object groups created at different times exhibit dramatically different mean reuse times. Using a group size of 100 objects on the same trace, Figure 6.3 shows that some groups exhibit more than $10\times$ higher mean reuse time than others. These high-reuse-time groups are potentially good candidates for eviction. The two observations illustrate the feasibility of group-level learning using write-time-based grouping: objects inside groups are similar. Grouping by write time also allows an efficient implementation using a log-structured cache.

6.2.4 Utility of object groups

Identifying a good eviction candidate in object-based eviction has been well-studied. When object size is uniform, Belady [39] algorithm evicts the object that is requested the furthest in the future. When object size is not uniform, identifying the optimal candidate is NP-hard [44]. A common approximation is to evict the object that has the largest time till the next request over object size (called "size-aware Belady"). However, no metric exists that applies to object groups, and it is not trivial to adapt object-level metrics to the group level. In this section, we define an *object-group utility* function to measure object-group usefulness. A group with a lower utility is less useful and hence should be preferred for eviction. Because identifying the optimal *object* for eviction (when objects do not have the same size) can be reduced to identifying the optimal *group* for eviction, and the former is NP-hard [44], finding the optimal group for eviction is also NP-hard. Therefore, we define an empirical group utility that satisfies several properties.

Desired properties

(1) Because large objects occupy more space, the utility should consider object sizes. Groups composed of larger objects should have lower utilities.

(2) Similar to Belady, the utility should consider the time till the next access of objects in the group. A group of objects that are requested further in the future should have a lower utility. Importantly, the utility definition should properly handle objects with no future requests.

(3) When the group size is one object, group-level learning becomes object-level learning. In this case, ranking using the defined utility should produce the same result as Belady.

(4) The utility should be easy and accurate to track online. Calculating the ground truth (used for training) requires future information, but the cache cannot wait indefinitely to

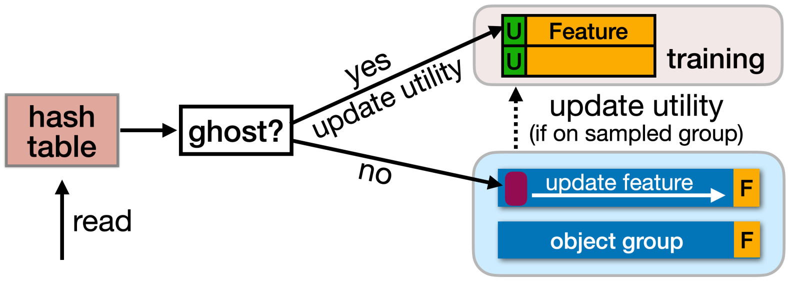


Figure 6.4: The read flow in GL-Cache.

calculate it. This property requires that within a limited time horizon, the online tracked utility should be close to the utility calculated with complete future information. In other words, objects requested further in the future, including the ones with no future requests, should contribute less to the utility.

Utility definition

We observe that the cost of evicting one object is *always only one miss*. After a cache miss, the evicted object will be inserted into the cache. Meanwhile, the benefit of evicting one object o is proportional to its size s_o and *time till next access* $T_o(t)$ from current time t. Therefore, similar to the cost-benefit analysis in LFS [292] and RAMCloud [264, 293], we define the utility of an object as its cost (one miss) over benefit (freed space multiplied by time till its next request).

$$U_o(t) = \frac{1}{T_o(t) \times s_o} \tag{6.1}$$

Because GL-Cache evicts object groups, we further define the group utility as the sum of object utilities.

$$U_{group}(t) = \sum_{o \in \text{group}} \frac{1}{T_o(t) \times s_o} \tag{6.2}$$

The utility of a group measures the penalty of evicting the group or the benefit of keeping the group. Groups with lower utilities are thus better candidates for eviction. We remark that this is one definition of group utility that both satisfies the desired properties and performs well in our experience (section 6.3). With this definition, we compare object-group utility and evict the group with the lowest utility. Since the true utility relies on the time till the next request and can only be calculated with future information, we design GL-Cache, which learns a model that can predict a group's utility based on its features.

6.2.5 Learning object-group utility in GL-Cache

GL-Cache learns a function $\mathcal{F}$ that calculates a group's utility given its features: $\mathcal{F}(X_{group}) = U_{group}$ where X_{group} is the features of an object group.

Object-group features. Features play a crucial role in learning [103, 147]. We consider two types of features in GL-Cache. The first type is *static features*, which includes request rate, write rate, miss ratio in the time window when the group was created (the write time of the first object), and mean object size. The second type is *dynamic features*, which includes age (in seconds), the number of requests, and the number of requested objects. Dynamic features increase over time. Static features do not change after creating a group and capture the workload and cache states (e.g., daily scan, request spike) during group creation time. We focus on these states because access pattern changes are often reflected in these metrics. For example, object groups created from scans are good candidates for evictions, and they often co-appear with increased request rates, write rates, and miss ratios. Compared to many of the existing works [312, 364], which mostly use dynamic features, GL-Cache uses far fewer dynamic features because tracking dynamic features is computationally expensive. We observe that adding more dynamic features only brings marginal hit ratio improvement, which does not justify the added computation overhead.

In total, GL-Cache uses seven features occupying 20 bytes for each group or 28 bytes if mean object size and creation time are not already tracked.

Learning model and objective function. GL-Cache uses gradient boosting machines (GBM) because tree models do not require feature normalization, and they have been shown to work well in previous works [42, 312] as well as many production environments [294, 338]. We formulate the learning task as a regression problem that minimizes the mean square loss (L2) of object-group utilities. We also explored the ranking objective function without observing a significant difference.

Training. GL-Cache trains a model using online collected training data, which consists of features and utilities of object groups. GL-Cache generates new training data by sampling cached object groups, and it copies the features of the sampled groups into a pre-allocated memory region. The utilities of the sampled groups are initialized to zero at the beginning and calculated over time. When an object o from a sampled group is requested, GL-Cache can calculate the $T_o(t)$ (time till next request since sampling) and object utility using Eq. 6.1 and add the object utility into the group utility. GL-Cache then marks the object to ensure that it only contributes to the group utility *once*. It is possible that some objects may not be requested before training, and the online calculated group utility may be lower than the true utility. However, as mentioned in subsection 6.2.4, these objects contribute marginally to the group utility due to their large reuse time.

In addition, a sampled group may be evicted before being used for training. Such evictions halt the tracking of group utility. Inspired by prior works [227, 291], GL-Cache keeps ghost entries for objects which have not been factored into group utility. A future request on the ghost entry will update the group utility, bringing it closer to the true utility.

Figure 6.4 shows the read flow in GL-Cache. A successful hash table lookup may find two types of entries: a pointer to the object or a ghost entry. If it is a regular object, GL-Cache first updates the group features. Further, if the object is on a sampled group and has not contributed to the group utility, GL-Cache also updates the group utility before returning the data to the user. If it is a ghost entry, GL-Cache updates the corresponding utility and removes the ghost entry from the hash table, then returns a cache miss.

Given the access patterns change over time, the model needs to be retrained regularly. GL-Cache retrains the model every day (i.e., using wall clock time as a reference) because many real-world events that trigger requests repeat on a daily basis, such as cron jobs. In contrast, the other option of retraining every certain number of requests may cause the system to enter metastable failure [55, 153] when an access pattern change increases the system load. Besides, GL-Cache chooses to retrain from scratch each time because tree models do not benefit from continuous training. Moreover, the inference overhead grows with training iterations because a new tree is added to the model in each iteration.

Inference. When GL-Cache needs to perform evictions, it predicts the utilities of all object groups and ranks them. GL-Cache uses the inference/ranking result for multiple evictions, which reduces the frequency of inference and thus the computation overhead. We denote *eviction fraction* $F_{eviction}$ as the fraction of ranked groups to evict using one inference. That is, GL-Cache performs an inference every $F_{eviction} \times N_{group}$ groups where N_{group} is the total number of groups. In our evaluation, $N_{ranked-group}$ is the total number of groups, but we remark that one can also sample some groups for inference if the total number of groups is too large. Also, the groups are evicted over time on demand rather than all at once, and neither training nor inference need to be on the critical path of request serving. In summary, GL-Cache only needs to perform $\frac{1}{F_{eviction}}$ inferences to write

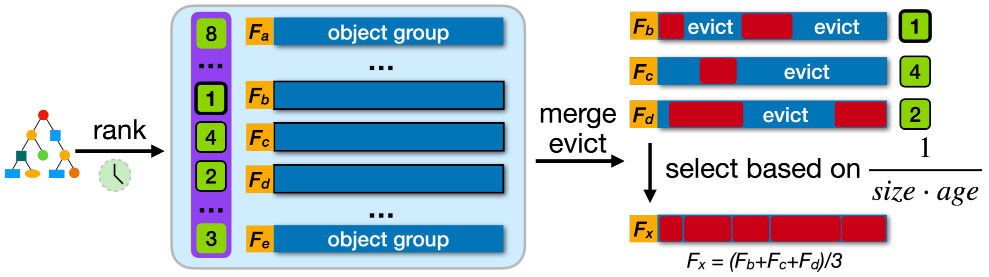


Figure 6.5: Object group utility prediction and merge-based group eviction in GL-Cache.

a full cache of objects.

6.2.6 Evictions of object groups

Learning at the object-group level introduces an interesting challenge to cache eviction: unlike most caches which evict one object each time, GL-Cache evicts a group of objects. Although evicting object groups leads to lower overhead due to batching and amortization, it may evict objects that are still popular. GL-Cache optimizes the group eviction by using a merge-based eviction, similar to Segcache [372]. Upon each eviction, GL-Cache picks the least useful object group and merges it with *the $N_{merge} - 1$ object groups that are closest with respect to write time*. The merge process retains S_{group} objects from the merged groups and evicts all other objects. The retained objects form a new group, and the original N_{merge} groups are evicted. This is the only time that an object changes its group membership in GL-Cache. Unlike group selection, which uses ranking, object selection uses a simple metric based on object age and size: $\frac{1}{size \cdot age}$ where age is the time since the last access. We choose to use this metric because recency and size are the two most common metrics used in other eviction algorithms (section 6.1). GL-Cache performs heavyweight online learning at the group level to identify the best groups to evict. It leverages lightweight object-level metrics to retain a few highly useful objects. This two-level eviction approach enables GL-Cache to achieve a superior tradeoff between learning overhead and cache efficiency.

In summary, each eviction evicts N_{merge} groups of objects and retains one group of objects, as illustrated in Figure 6.5. The features (except mean object size) of the merge-produced group take the mean values of the N_{merge} merged groups. Note that only the first object group is picked based on the group utility; the next $N_{merge} - 1$ object groups are chosen as ones with write time close to the first group. This ensures that objects

Table 6.2: Parameters used in the design.

Parameter	Meaning
S_{group}	Size of an object group (in number of objects or bytes)
N_{merge}	Number of object groups to merge each eviction
$F_{eviction}$	Each inference evicts $F_{eviction}$ fraction of ranked groups

in the new group after a merge-based eviction are still close in write time and similar. In contrast, objects from the N_{merge} least useful groups may not be similar. Clustering similar objects into groups is critical for effective group-level learning. In our experience, merging the N_{merge} least useful groups shows lower efficiency with up to 20% decrease in hit ratio.

Compared to evicting one object each time, group-based eviction evicts more objects than needed at each eviction, which may reduce the efficiency upper bound group-level learning can achieve. However, we show in subsection 6.3.2 that evicting object groups can achieve hit ratios very close to Belady, indicating that group eviction will not be the bottleneck for cache efficiency.

6.2.7 A spectrum of GL-Cache

GL-Cache has three parameters in its design (Table 6.2): the size of each object group S_{group}, the number of object groups to merge at each eviction N_{merge}, and how many groups are evicted using one inference which is determined by $F_{eviction}$. Varying these parameters leads to a spectrum of algorithms for optimizing hit ratio and throughput. A larger S_{group} reduces learning granularity; a larger N_{merge} retains fewer objects; and a larger $F_{eviction}$ reduces the ranking frequency. Each of these changes reduces the computation overhead with a potential hit ratio drop. Therefore, GL-Cache allows the users to navigate the trade-off between cache efficiency and throughput. For scenarios that are more sensitive to overheads, such as local cache deployments, GL-Cache can provide higher throughput with a slightly lower hit ratio, and vice versa. In subsection 6.3.6, we show that these parameters generalize well across workloads.

Table 6.3: Three sets of 128 traces were used in the evaluation.

Dataset	# traces	# requests (millions)	Source
CloudPhysics [333]	103	2115	VM disk I/O
MSR [252]	14	410	Disk I/O
Wikimedia [312]	1	2804	CDN requests

6.3 Evaluation

In this section, we evaluate GL-Cache to answer the following questions.

- Will group-based eviction limit the efficiency upper bound when compared to object-based eviction (subsection 6.3.2)?
- Can GL-Cache improve hit ratio and efficiency over other learned caches (subsection 6.3.3)?
- Can GL-Cache meet production-level throughput requirements and how much overhead does GL-Cache add (subsection 6.3.4)?
- How does GL-Cache improve efficiency without compromising throughput (subsection 6.3.5)?

6.3.1 Experiment methodology

Prototype system. GL-Cache groups objects using write time and can be efficiently implemented using a log-structured cache. Hence, we implement GL-Cache on top of Segcache [372], an open-source production in-memory cache that uses segment-structured (log-structured) storage. We map an object group in GL-Cache to a "segment" in Segcache and replace FIFO with the learned model. We use the XGBoost [358] library to implement our GBM models and use the default values for all parameters. GL-Cache has three parameters (Table 6.2). In our evaluation, GL-Cache uses 1 MB group size, merges five groups at each eviction, and evicts 5% of ranked groups after each inference. We compare GL-Cache with Segcache [372], a segment-structured cache used by Twitter; Cachelib [41], Meta's production cache library, which uses slab storage and a throughput-optimized LRU for eviction; TinyLFU [109], implemented within Cachelib by Meta engineers. We have also implemented LHD [38] on top of Pelikan's slab storage [97].

Micro-implementation. In addition to the prototype system, we build a storage-oblivious implementation of GL-Cache in C on top of libCacheSim [368] to compare different eviction algorithms. Our implementation mimics Memcached's design but has neither a networking stack nor object value storage, and we call it micro-implementation. Compared to the prototype, the micro-implementation only performs eviction-related metadata operations and does not consider storage layout or system overheads such as fragmentation. We use two sets of parameters (Table 6.2) to demonstrate the spectrum of GL-Cache. The first demonstrates a better *efficiency* and uses $S_{group} = 60$ objects, $N_{merge} = 2$ groups, $F_{eviction} = 0.02$. We call this system GL-Cache-E. The second demonstrates a higher *throughput* using $S_{group} = 200$ objects, $N_{merge} = 5$ groups and $F_{eviction} = 0.1$, and we call it *GL-Cache-T*. We remark that the parameters are not tuned per workload. Thus GL-Cache may provide better performance (hit ratio or throughput) with workload-specific fine-tuning.

Besides GL-Cache, we implement Cacheus [291] in C following the authors' open-source Python implementation. For LHD [38] and LRB [312], our micro-implementation used code open-sourced by the authors. We use default parameters except for changing the LRB optimization target from byte miss ratio to object miss ratio (implemented by LRB's author). Besides state-of-the-art designs, we have also implemented FIFO, LRU, and size-aware Belady [44].

GL-Cache trains the first model after running one day of workload (using timestamps from the traces). Before a model is trained, it uses FIFO to perform evictions, GL-Cache then trains the model once a day from scratch, which has little overhead as discussed in subsection 6.2.5.

Workloads. We use a wide variety of traces representing a diverse set of workloads from three dataset sources (Table 6.3). The CloudPhysics [333] dataset includes 103 block I/O traces with different CPU/DRAM configurations and access patterns. Each trace records the I/O requests from a VM for around one week. Because 86% of the VMs had DRAM sizes between 1 GB and 16 GB with a median of 3880 MB, we performed evaluations at 1 GB, 4GB, and 16 GB cache sizes. We present only 1 GB and 16 GB for space reasons. We have also evaluated GL-Cache using 14 block I/O traces (we ignore the traces which contain fewer than 5 million requests) from Microsoft Research Cambridge (MSR) [252]. Because the working set sizes of MSR traces exhibit a very wide range, we set cache sizes for each trace at 0.01%, 0.1%, and 1% of each trace's footprint (size of all objects). Besides block I/O request traces, we have also evaluated GL-Cache with the Wikimedia CDN trace used in previous works such as LRB [312] and LFO [42]. All the workload traces have at least three fields: the timestamp, id, and size of the requests.

We ran micro-implementation experiments on the Cloudlab [107] Utah site using m510 nodes with Intel Xeon D-1548 CPU, 64GB ECC DDR4 DRAM. And we ran prototype experiments on the Cloudlab Clemson site using c6420 nodes with Intel Xeon Gold 6142 CPU and 384 GB of DRAM.

Metrics. We replayed traces by reading and writing to a local cache in a closed loop and measured hit ratio and throughput. Because all traces are week-long traces, we started measurements after finishing the first three days' requests to make sure the cache is properly warmed up under all the configurations considered. We present evaluations using a one-day warmup time in subsection 6.3.6, which shows that the observations remain the same as with a three-day warmup.

We report aggregated results from 103 CloudPhysics traces and 14 MSR traces using box plots for the micro-implementation results. Due to the diversity of the workloads, both hit ratio and throughput have wide ranges. Hence, for ease of visual presentation, we report results compared to FIFO using the following two metrics: *hit ratio increase over FIFO* defined as $\frac{HR_{alg}-HR_{FIFO}}{HR_{FIFO}}$ where HR stands for hit ratio; throughput relative to FIFO defined as $\frac{R_{alg}}{R_{FIFO}}$ where R is the throughput. The box plots have the following format: the orange line inside the box is the median, the box shows 25 and 75 percentiles, and the whiskers show 10 and 90 percentiles. Because several other factors in the prototype systems (e.g., storage layout) affect efficiency and throughput, for ease of understanding, we focus our evaluation on the micro-implementation results. We present raw hit ratio and throughput numbers using the prototype systems for one representative trace in subsection 6.3.3 and subsection 6.3.4.

6.3.2 Group-based eviction

Group-level learning evicts most objects in the selected groups. The bulk eviction may limit the efficiency of group-level learning. To understand the limitation of group eviction, we compare oracle-assisted group eviction with oracle-assisted object eviction (size-aware Belady [44]). The oracle-assisted group eviction uses the same design as GL-Cache except

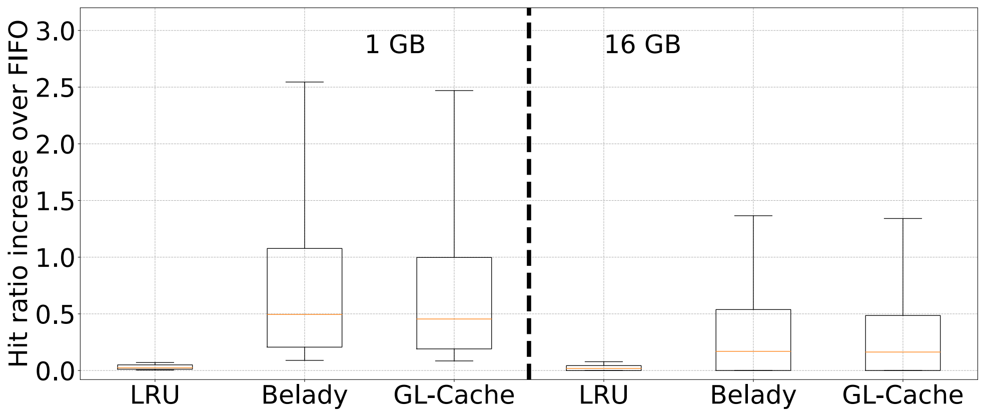
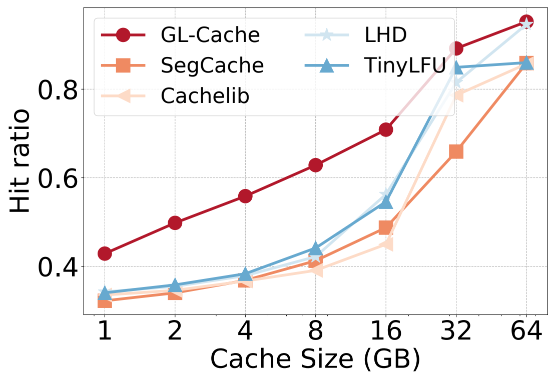
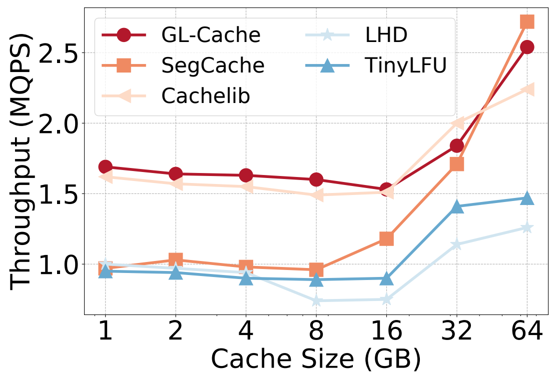


Figure 6.6: With oracle assistance, group eviction can achieve a similar hit ratio improvement as object eviction.



(**a**) Hit ratio

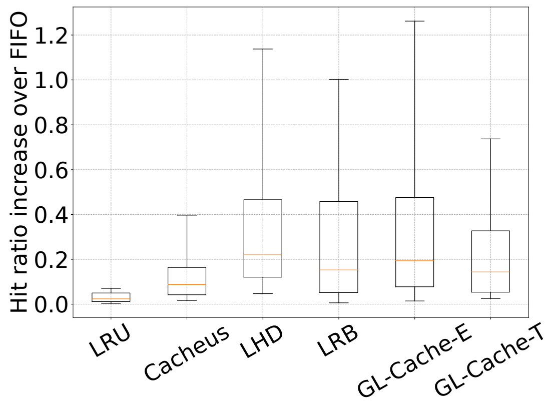


(**b**) Throughput

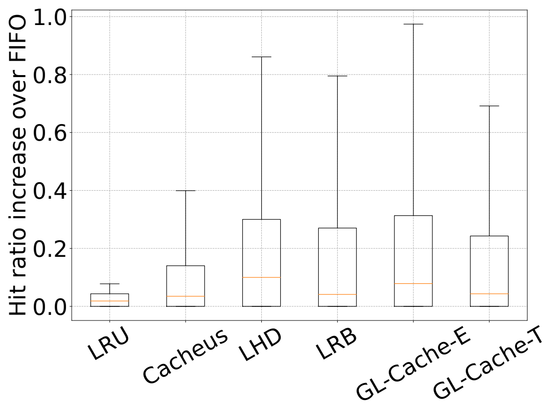
Figure 6.7: Prototype evaluation of a CloudPhysics trace.

using future request time to calculate group utility and retain objects. Size-aware Belady evicts the object that has the largest $(T_{next} - T_{now}) \times s_o$ where T_{next} is the time of the next request, and s_o is the object size.

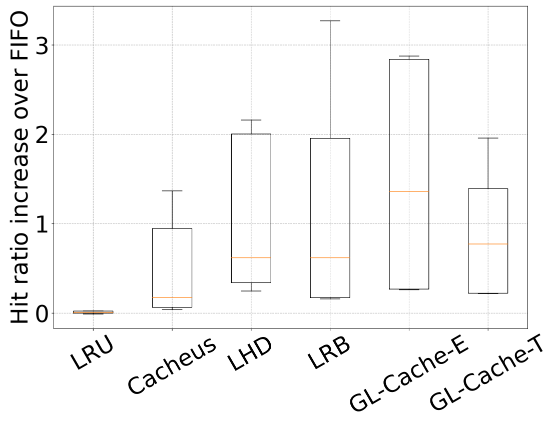
We compare these two approaches using CloudPhysics traces. Figure 6.6 shows that group-based eviction can achieve a hit ratio similar to object-based eviction at both small and large cache sizes. The similar hit ratios suggest that *group eviction will not become the bottleneck for achieving high efficiency*. While the algorithms in this comparison use oracle information, in the following sections, we show how GL-Cache can use learning to replace the oracle and achieve high cache efficiency.



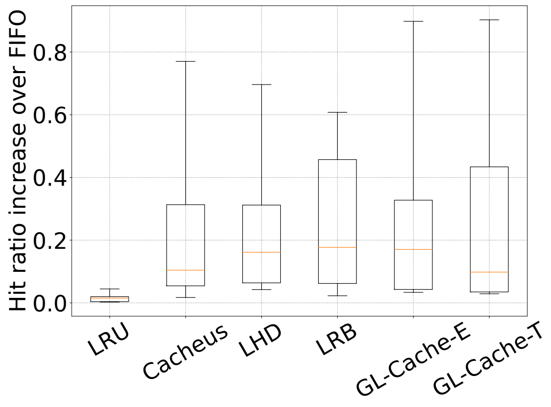
(**a**) CloudPhysics, small cache size



(**b**) CloudPhysics, large cache size



(**c**) MSR, small cache size



(**d**) MSR, large cache size

Figure 6.8: Hit ratio increase over FIFO. GL-Cache runs under two modes, GL-Cache-E is the efficient mode, GL-Cache-T is the throughput mode.

6.3.3 Cache efficiency

We compare the efficiency of GL-Cache with state-of-the-art designs in both the prototype and the micro-implementation. Figure 6.7a shows hit ratios for the prototype running one CloudPhysics trace at different sizes. Compared to other systems, GL-Cache consistently achieves the best efficiency, providing a significant hit ratio increase (up to 40%) over the best of all baselines. Compared to Segcache, which uses the same storage layout with FIFO-based group eviction, group-level learning increases the hit ratio by 60% at 8 GB. Cachelib uses a throughput-optimized LRU and has the lowest hit ratio among all the baselines. LHD and TinyLFU use two object features to make eviction decisions: LHD models hit density based on age and size; TinyLFU uses frequency to filter out unpopular objects and uses recency to evict objects. Leveraging more than one feature to choose eviction candidates allows LHD and TinyLFU to achieve higher hit ratios. However, not using more features puts an upper bound on their potential. In comparison, GL-Cache evicts groups based on seven features covering recency, frequency, cache, and workload states at group creation time (miss ratio, write rate, request rate). Considering multiple features in conjunction with learned importance allows GL-Cache to make better eviction decisions and achieves a higher hit ratio. Evaluations on the other traces show similar results.

To compare with more algorithms and on more traces, we show hit ratio results from the micro-implementation on CloudPhysics and MSR traces in Figure 6.8. Because of the wide range of hit ratios across traces, we show the relative hit ratio increase compared to FIFO instead of the raw hit ratios. We observe that both LRU and Cacheus improve FIFO's hit ratio, but only by a single-digit percentage for the median workload on both datasets. Meanwhile, LRB, LHD, and GL-Cache increase FIFO's hit ratio more prominently.

Among LRB, LHD, and GL-Cache-E, LRB has the smallest observed hit ratio improvement. We conjecture that learning at the object level receives limited information on each object since cache workloads often follow Zipf distributions, and thus is more challenging to learn compared to learning at the group level. Compared to LHD, we observe that GL-Cache-E shows similar efficiency on CloudPhysics traces. However, on MSR traces, GL-Cache-E is more efficient than LHD with a 60% hit ratio increase for a median workload at the small size. This observation suggests that leveraging more features to make eviction decisions can be very useful for some workloads at certain cache configurations.

Compared to GL-Cache-E, GL-Cache-T trades hit ratio for higher throughput (subsection 6.3.4). However, we observe that GL-Cache-T's efficiency is still on-par with LRB. Overall, we observe that GL-Cache improves the hit ratio by up to 37.8% compared to LHD and 87% compared to LRB (not shown in the figure). While LRB uses more features/information than other eviction algorithms, it does not always provide the highest hit ratio. More information leads to higher efficiency only when the information is useful and well-utilized. We conjecture that perhaps not all the features in LRB are useful, and the model may not be making the best use of the features.

When comparing prototype and micro-implementation results, we observe that the hit ratio difference also depends on the storage design. GL-Cache uses log-structured

Table 6.4: Comparing LRB and GL-Cache-E on the Wikimedia trace used in LRB paper [312]. We use miss ratio because it is more commonly used in web caches.

Algorithm	Miss ratio			Throughput (MQPS)		
Size (GB)	20	200	2000	20	200	2000
FIFO	0.39	0.16	0.025	7.62	7.91	9.68
LRB	0.24	0.048	0.016	0.01	0.04	0.07
GL-Cache-T	0.24	0.065	0.017	4.97	6.53	4.89
GL-Cache-E	**0.20**	**0.041**	**0.013**	2.55	3.91	4.20

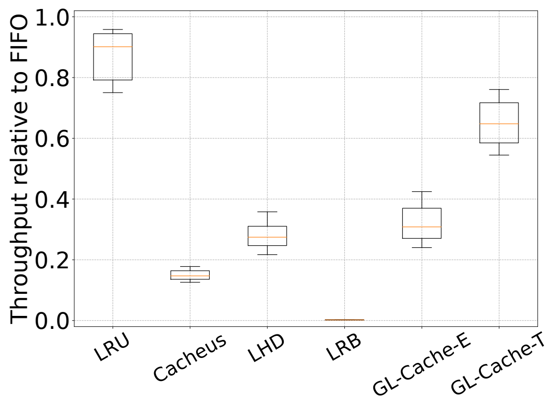
storage, and the difference between prototype and micro-implementation is smaller (<10%); LHD uses slab storage, and sometimes the prototype can have a significantly lower hit ratio (>20%) compared to the micro-implementation. This large difference comes from fragmentation and slab calcification problems [151, 372]. However, we did not find a way to efficiently implement LHD on top of log-structured storage because it requires the storage to have the capability of evicting (removing) any cached object, while log-structured storage can only efficiently support sequential write and removal.

Besides block I/O cache traces, we have also evaluated GL-Cache using the Wikimedia CDN trace from LRB evaluations. Table 6.4 shows that learning helps LRB to achieve miss ratios 35% to 70% lower than FIFO. Compared to LRB, GL-Cache-E further reduces the miss ratio by up to 16%. In summary, the evaluations on three datasets totaled 118 traces illustrating the high efficiency and generality of group-level learning.

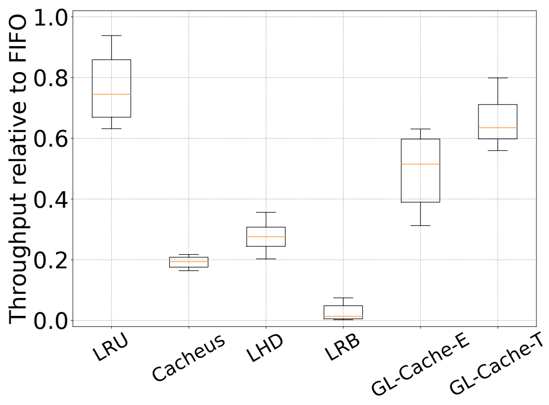
6.3.4 Throughput and overheads

Not only does GL-Cache achieve a high hit ratio, but GL-Cache also achieves high throughput. Figure 6.7b shows the throughput of GL-Cache in the prototype. We observe that compared to production systems (Cachelib, Segcache), GL-Cache achieves a similar throughput, indicating that GL-Cache meets the throughput requirement of a production system. Moreover, compared with eviction algorithms such as LHD and TinyLFU, GL-Cache is 2-3$\times$ faster.

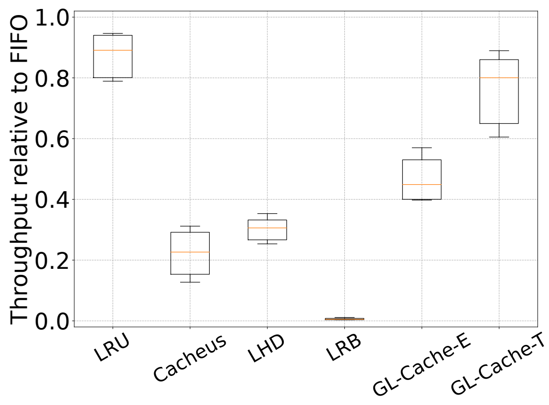
Besides the prototype evaluation, Figure 6.9 compares the throughput of GL-Cache with several state-of-the-art algorithms evaluated on *all* CloudPhysics and MSR traces. While LRU achieves throughput close to FIFO, all advanced eviction algorithms exhibit a



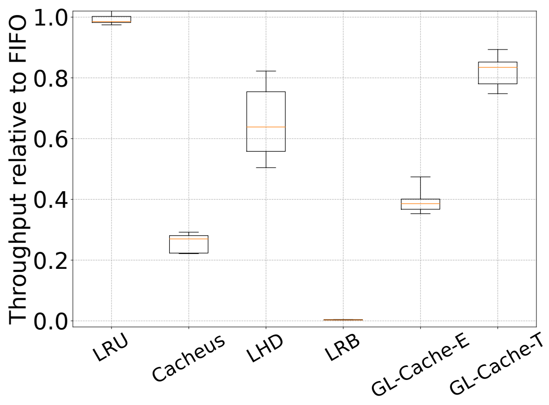
(**a**) CloudPhysics, small cache size



(**b**) CloudPhysics, large cache size



(**c**) MSR, small cache size



(**d**) MSR, large cache size

Figure 6.9: Throughput relative to FIFO.

significant slowdown compared to FIFO. However, among all learned caches, GL-Cache is significantly faster than others. Compared to LRB, GL-Cache-E has a 228× higher throughput, and GL-Cache-T has a 586× higher throughput on average at the small cache size. Compared to the *fastest* of all learned caches, GL-Cache-E is on average 64% faster, and GL-Cache-T is on average 3× faster at the small cache size. Similarly, on the Wikimedia trace (Table 6.4), GL-Cache-E is tens to hundreds of times faster than LRB and achieves almost half of FIFO's throughput.

GL-Cache achieves high throughput because it needs very few metadata updates on cache hits and misses. On a cache hit, GL-Cache only needs to update the last access time and group utility if it is on a sampled group (subsection 6.2.5). On a cache miss, GL-Cache does not need to update any metadata most of the time; occasionally, it performs a group eviction and evicts 100s to 1000s of objects. In contrast, other systems must

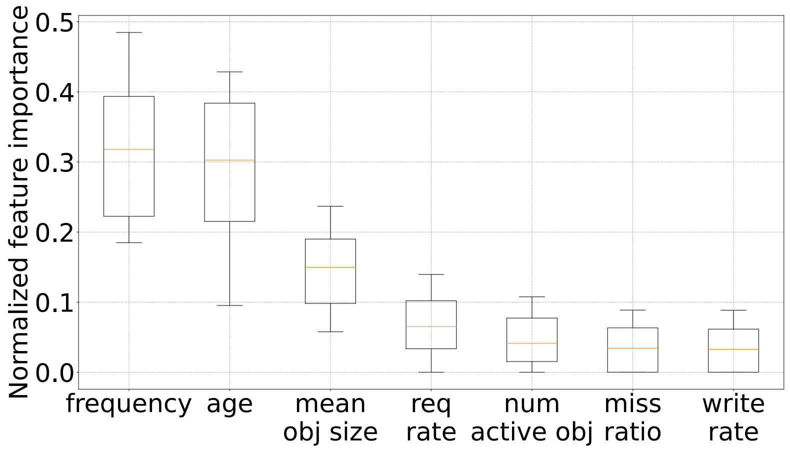


Figure 6.10: Feature importance across traces.

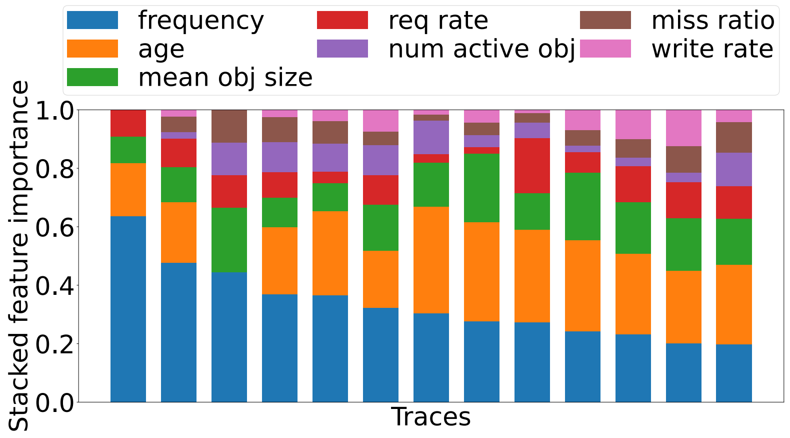


Figure 6.11: Feature breakdown across example traces.

update multiple metadata entries on both cache hits and cache misses. For example, TinyLFU needs to maintain the frequency counting sketch and the LRU chain; LHD needs to sample 32 objects, thus having 32 random DRAM accesses for each eviction. Segcache is simpler than GL-Cache in per-request operations. However, the lower hit ratio of Segcache leads to its reduced throughput because of more evictions.

The second reason for GL-Cache's high throughput is that the overheads of training and inference are amortized. Because GL-Cache uses fewer features to learn simpler high-level patterns instead of per-object access patterns, it uses a simple model and is only retrained once a day. In our measurement, each training consumes 10 - 50 ms of one CPU core (not amortized by the number of training samples). In addition, each inference consumes 0.4 - 3 ms of one CPU core and is triggered every time 5% of ranked groups

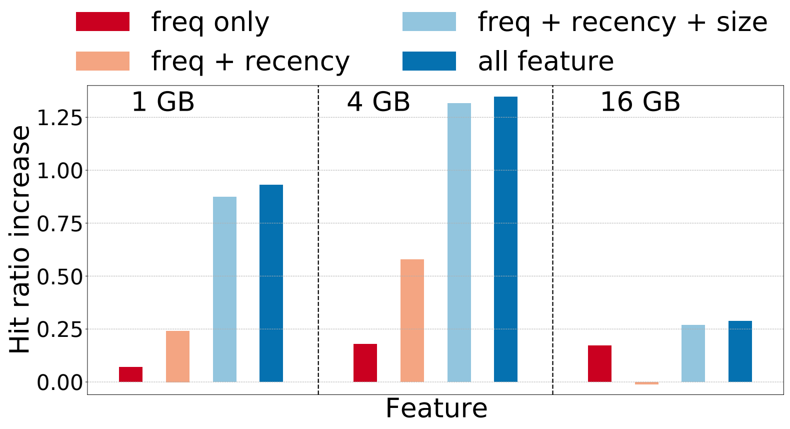


Figure 6.12: Feature utility case study.

are evicted. Because each inference evicts many groups and each eviction evicts many objects, the inference computation is amortized. The amortization is the key reason for GL-Cache's high throughput compared to other learned caches. Moreover, although training and inference are not on the critical path of request serving, our throughput evaluation measures run time including both training and inference.

While throughput evaluations show the low computation overhead of GL-Cache, machine learning in caching also introduces storage overhead. First, GL-Cache uses DRAM to store 8000 training samples. The training data storage is pre-allocated and small (256 KB) compared to the cache size (GBs). For deployments with very limited memory, the training data can also be stored on the storage device. Second, each object group in GL-Cache uses 28 bytes of features — each object thus adds less than one byte. Besides the group-level features, GL-Cache tracks each object's last access time using 4 bytes. In total, GL-Cache uses 5 bytes of object metadata for eviction. As a comparison, LRU requires two pointers with 16 bytes of metadata per object, and LRB uses 192 bytes of features per object.

6.3.5 Understanding GL-Cache's efficiency

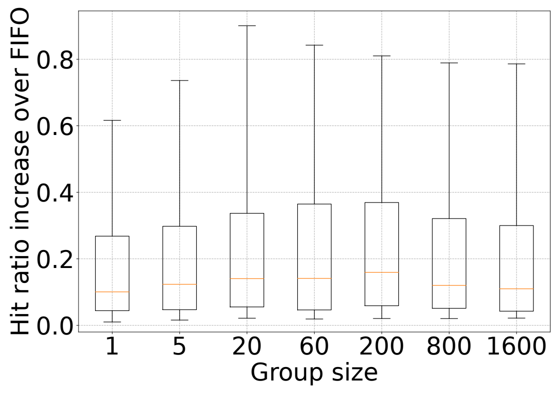
So far we have demonstrated that GL-Cache has a higher miss ratio and throughput than existing systems. While amortized overhead explains the high throughput, this section explores how learning helps GL-Cache achieve high efficiency.

Most eviction algorithms use one or two object features to decide which object to evict. For example, LRU evicts the object with the largest access age (recency), LeCaR and Hyperbolic [48] use recency and frequency to make eviction decisions, LHD relies on access age and object size to choose eviction candidates. In contrast, object-level learned cache such as LRB uses 44 features covering different measurements of recency and frequency, as well as object size, to compare objects. Similarly, GL-Cache uses seven features to compare object groups. To better understand GL-Cache's efficiency, we examine how GL-Cache uses these features.

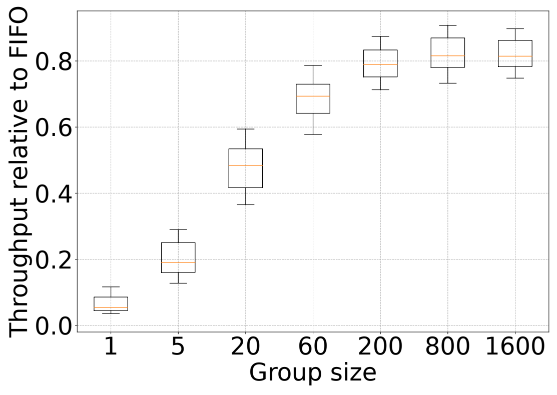
We obtained the feature importance score directly from XGBoost. The importance score is calculated using the number of times a feature is used to split the data across all trees and may not represent the ground truth. Figure 6.10 shows the normalized feature importance scores of different features across traces obtained from the models trained for each trace. We observe that across traces, frequency and age have relatively high scores with medians of around 0.3. This aligns well with existing literature on eviction algorithms, which mostly use recency and frequency to make eviction decisions. The next important feature is the mean object size, which is essential for algorithms that consider variable-size objects. Besides these features, the workload and cache states (request rate, miss ratio, write rate) at the group creation time have similar scores with a median of around 0.05. When summed up, they have a similar importance as the object size.

While we observe that the most commonly used features (recency, frequency, size) are critical, we also observe that no feature is dominant across all traces. Figure 6.11 shows the feature importance score for 12 randomly selected traces. For some traces, frequency is more important, with an importance score of 0.6. For others, recency or size is more important. GL-Cache weighing features differently across traces suggests that GL-Cache can effectively adapt the feature importance to each workload. For comparison, the algorithms leveraging more than one feature often combine the features in a way that cannot adapt to workloads. For example, Hyperbolic scores an object using $\frac{frequency}{age}$, leaving the *relative* importance of frequency and age unchanged across workloads.

Figure 6.12 uses one trace to illustrate the importance of GL-Cache *adaptively* using multiple features. It shows how gradually including more features improves the hit ratio. We observe that the combination of frequency, recency, and size at small sizes (1 GB and 4 GB) leads to a large hit ratio increase (e.g., 80% at 1 GB). Meanwhile, frequency alone is insufficient and can only increase the hit ratio by 10% at 1 GB. Using all features increases the hit ratio modestly on this trace compared to only using frequency, age,



(**a**) Hit ratio



(**b**) Throughput

Figure 6.13: Impact of group size.

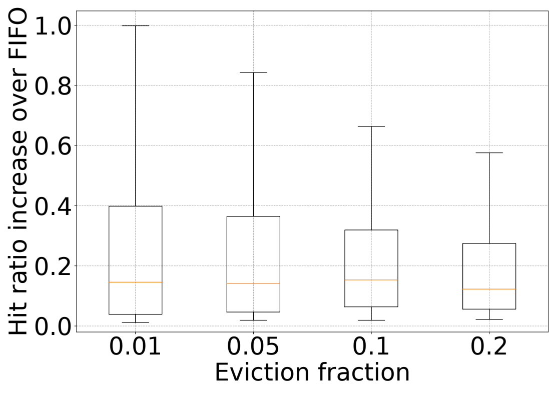
and size. Moreover, Figure 6.12 shows that feature importance could change with cache sizes. Object size is more important at 1 GB cache size, while frequency becomes more important than other features at 16 GB. This could be because small objects contribute more hits per consumed byte than large objects, so caching small objects is better when the cache size is small. Meanwhile, when most small objects are cached at a larger cache size, choosing between large objects depends on request frequency. This observation suggests that in GL-Cache, the choice and use of features adapt not only to the workloads but also to different configurations such as cache sizes.

In summary, learning at the group level can leverage multiple features to adapt to both workload and cache sizes, enabling higher cache efficiency.

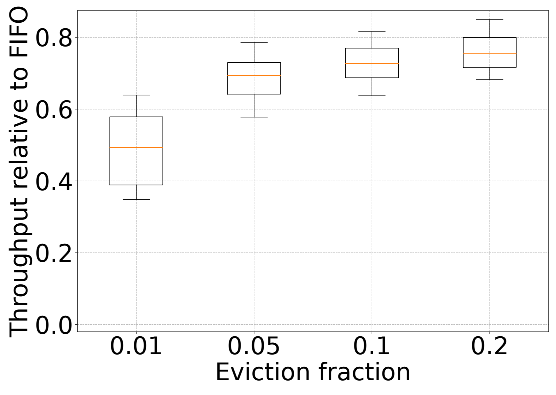
6.3.6 Sensitivity analysis

We have discussed the three parameters used by GL-Cache in subsection 6.2.7, and we have shown the two modes of GL-Cache: one achieves higher efficiency (GL-Cache-E), and the other achieves higher throughput (GL-Cache-T). This section shows in detail how these parameters affect hit ratio and throughput. In addition, we show that the warmup time does not significantly change the hit ratios.

Group size. A smaller group indicates a finer granularity for learning and evictions. Varying group size affects both throughput and efficiency. First, reducing group size increases storage and computation overhead due to finer learning granularity. As a result, throughput increases with group size, as shown in Figure 6.13. Second, the hit



(**a**) Hit ratio



(**b**) Throughput

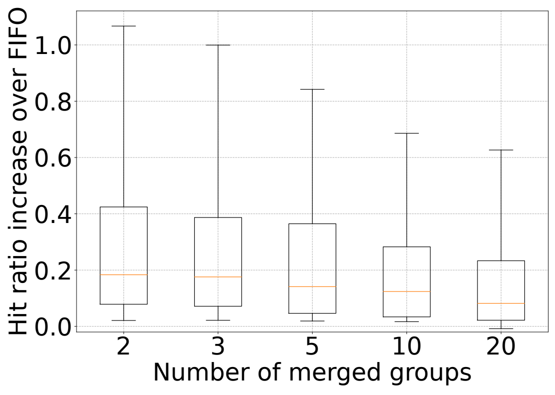
Figure 6.14: Impact of eviction fraction $F_{eviction}$.

ratio increases when the group size increases from 1 (object-level learning) to 20, then decreases as the group size further increases from 60 to 1600. A smaller group indicates that each eviction evicts fewer objects, enabling a higher hit ratio. However, when the group size is too small, each group gets too few requests for group feature learning to be effective, thus decreasing the hit ratio. The non-monotonic hit ratio change (hit ratio first increases then decreases) also explains why object-level learning achieves a lower hit ratio than GL-Cache.

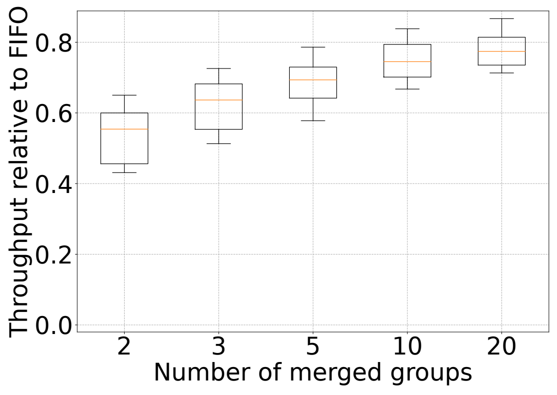
Eviction Fraction. GL-Cache evicts $F_{eviction}$ fraction of ranked groups between each inference to reduce computation overhead and better tolerate inaccurate predictions. The more groups (larger $F_{eviction}$) evicted per inference, the fewer inferences, thus higher throughput. However, a larger $F_{eviction}$ means more (useful) groups are evicted after each inference, resulting in a lower hit ratio. Figure 6.14 shows that increasing $F_{eviction}$ reduces hit ratio and increases throughput.

Number of groups to merge. The last tunable parameter in GL-Cache is the number of groups to merge at each eviction. Because GL-Cache evicts the majority of the objects on the N_{merge} groups and retains one group worth of objects, merging more groups means that GL-Cache retains fewer objects from each group. Retaining fewer objects reduces the computation needed at each eviction, but it also reduces efficiency. Figure 6.15 shows that increasing the number of merged groups increases throughput and reduces the hit ratio.

Besides the above three parameters, the learning component also introduces several



(**a**) Hit ratio

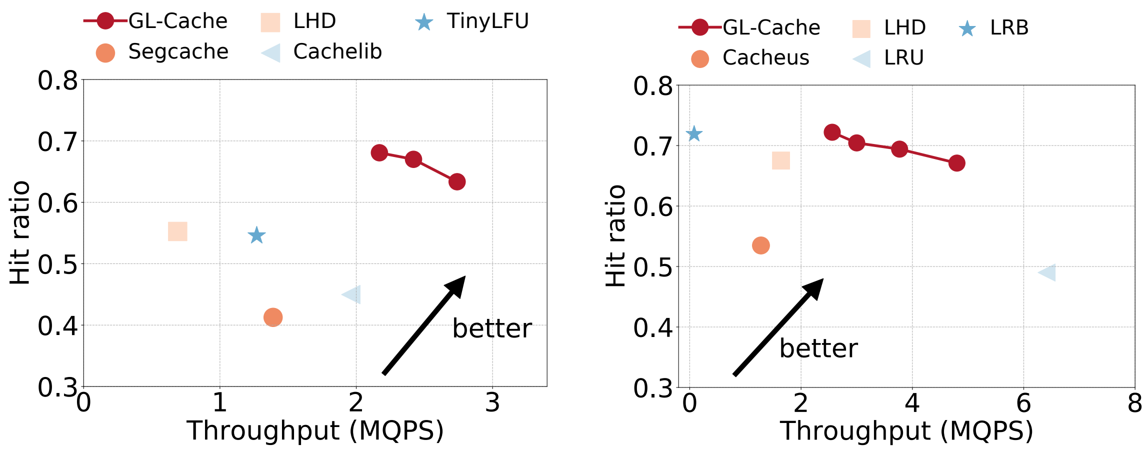


(**b**) Throughput

Figure 6.15: Impact of the number of groups to merge at each eviction.

parameters, such as training data size and retraining frequency. GL-Cache retrains the model once a day because many events (such as cron jobs and diurnal patterns) happen on a daily basis. Wall clock time sometimes is more important than virtual time (reference count) and has also been recognized by researchers from Google when they use neural networks to predict the lifetime of a memory allocation [217]. The retraining interval affects both efficiency and performance. Note that more frequent retraining does not always lead to a higher hit ratio because shorter retraining intervals reduce the accuracy of the group utilities used for training as they are accumulated over time. We observe that the best retraining interval depends on the workload — some workloads show higher hit ratios with half-day retraining, and some others benefit from two-day retraining. While fine-tuning retraining intervals can improve the hit ratio by up to 10%, one-day retraining achieves a good performance across workloads as shown. Besides training frequency, another parameter in training is the number of training samples. Because GL-Cache learns high-level access patterns, which we conjecture is easier to learn than per-object behavior, GL-Cache does not require a large amount of training data. While we cannot prove that 8000 training samples are sufficient for all workloads under all scenarios, we find that it is sufficient for the diverse traces in our evaluation.

The sensitivity analysis shows that GL-Cache is relatively robust to parameter changes. The parameters of GL-Cache-E and GL-Cache-T were chosen based on evaluations of 10 random traces. Our results show that these two sets of parameters work well across the diverse traces in the evaluation. However, like in any other system, a general set of parameters provides reasonable performance but does not guarantee the best performance.

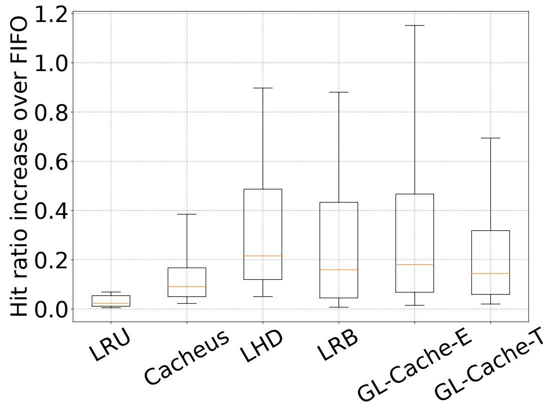


(a) Prototype evaluation (b) Micro-implementation

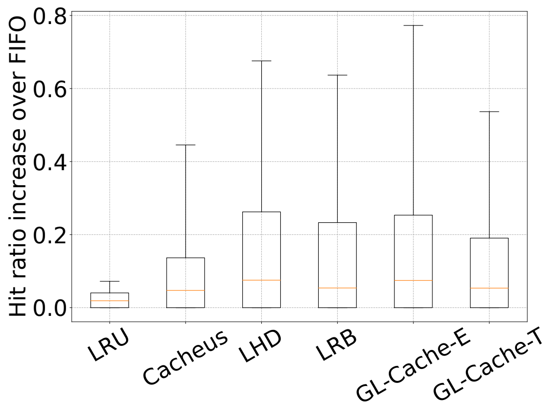
Figure 6.16: A spectrum of GL-Caches allow users to tradeoff between hit ratio and throughput.

Per-workload fine-tuning can potentially provide larger benefits. GL-Cache provides the opportunity for users to explore the trade-off between efficiency and throughput. Figure 6.16 shows the throughput and hit ratio of GL-Cache compared to baselines (we do not plot multiple close-by points of GL-Cache for clarity). In both prototype and micro-implementation evaluations, GL-Cache achieves higher throughput than systems with a similar hit ratio or a higher hit ratio than systems with a similar throughput. Deployments with less computation power can use GL-Cache in a high-throughput mode with a slightly lower hit ratio. And deployments that are less sensitive to computation may use GL-Cache to achieve a higher hit ratio.

Our evaluation so far used a warmup time of three days to make sure the cache is warmed up for any trace under any size. We have also evaluated with a one-day warmup time and presented the results in Figure 6.17. We observe that although the absolute values exhibit some differences, the overall trends on hit ratio increase are similar when compared to using a three-day warmup time (Figure 6.8). In addition to the hit ratio results, throughput results using a one-day warmup are also similar to that of a three-day warmup. Similarly, evaluations on the MSR and Wikimedia traces also exhibit little difference between using one-day and three-day warmup times.



(**a**) Small cache size



(**b**) Large cache size

Figure 6.17: Using one-day warmup, evaluated on CloudPhysics traces.

6.4 Related work

The study of cache designs has a long history with the majority of works focusing on improving cache efficiency. With increasing complexity in cache management, many recent works have also improved the throughput and scalability.

Better eviction algorithms. Most works improving cache efficiency focus on cache eviction algorithms, especially how to define and use recency, frequency, and size to make better eviction decisions. For example, ARC [227] uses two LRU lists to balance between recency and frequency; CAR [33], LIRS [165, 191, 393], Clock-pro [166], 2Q [172], SLRU [154], LRU-K [263] use a different metric to measure recency; variants of LFU [23, 176], LRFU [105], tinyLFU [108, 109, 110, 111] and hyperbolic [48] use a combination of frequency and recency to make evictions; various greedy-dual algorithms [82, 193, 392] use two metrics (e.g., frequency and size) to choose eviction candidates. In addition, several learned caches have been designed in the past few years, as discussed in detail in subsection 6.1.1. Compared to existing learned caches, GL-Cache employs group-level learning, which amortizes overheads and accumulates stronger learning signals to make better eviction decisions. Moreover, existing learning approaches to caching cannot be directly applied to group-level learning due to challenges such as comparing object groups' usefulness.

Improve cache throughput. Most algorithms that improve efficiency trade throughput for higher efficiency. With increasing complexity in cache systems, throughput and scalability become critical. MICA [203] uses a holistic design with a lossy hash table and partitioned log-structured DRAM storage to achieve high throughput and scalability; Segcache [372] uses an approximate-TTL-indexed segment-chain with batched eviction to achieve high throughput and scalability; MemC3 [118] uses a cuckoo hash table and Clock eviction to improve scalability; Cachelib [41] reduces LRU promotion frequency to improve scalability. These systems often use weaker eviction algorithms such as FIFO, Clock, or weak LRU. Compared to these works, GL-Cache improves efficiency without sacrificing throughput. Specifically, GL-Cache and Segcache share some design aspects such as object grouping. However, Segcache primarily innovates on the design of storage layout for key-value caches, and it uses FIFO for eviction. Instead, GL-Cache focuses on using learning for evictions, which is the key to GL-Cache's efficiency gains.

Use of machine learning to improve system efficiency. Machine learning has seen increasing use to improve system efficiency. For example, Google uses machine learning to improve the efficiency of data center operations [127]. Microsoft uses machine learning to improve database query optimizer [171]. Prior works have designed learned components to replace various parts of a system, such as index [100, 101, 185, 254] and query optimizer [221, 222] in databases, straggler mitigation in inference systems [183, 184], and FTL for SSD [316]. Moreover, many other works look into automatic database tuning using machine learning [194, 327]. In caching, in addition to the three categories of learned cache evictions that we have discussed in section 6.1, recent works have also looked into using sub-sampling to reduce learned cache's time horizon [334], using machine learning to predict memory access [146], designing cache admission [138, 181], designing cache prefetching [201, 314, 369] predicting hot records in LSM-Tree storage [377], using deep recurrent neural network models for content caching [251], using Markov cache model for size-aware cache admission policy [43]. Compared to these works, GL-Cache is the first system to perform learning on a group of entities and navigates efficiency-throughput trade-offs using coarse-grained learning granularity.

6.5 Chapter Summary

This chapter introduces three types of learned caches: *"object-level learning"*, *"learning-from-distribution"*, and *"learning-from-simple-experts"*. One of the biggest challenges of deploying a learned cache is the overhead.

This chapter then presents **G**roup-level **L**earned **Cache** (GL-Cache), which leverages group-level learning to overcome these challenges. GL-Cache clusters similar objects into groups using write time and evicts the least useful groups using a merge-based eviction. GL-Cache introduces a group utility function to rank groups, which enables group-based eviction to achieve efficiency similar to object-based eviction. GL-Cache uses a hybrid approach for eviction: it performs the heavyweight learning at the group level (thus amortizing the overheads) to identify the best groups to evict. And it leverages lightweight object-level metrics to retain a few highly useful objects from evicted groups. This two-level eviction enables GL-Cache to achieve a superior trade-off between learning overhead and cache efficiency.

We implemented GL-Cache in an open-source production cache and also developed a storage-oblivious implementation for running microbenchmarks. We compare GL-Cache with state-of-the-art designs on 118 production block I/O and CDN cache traces. Compared to object-level learning (LRB), group-level learning allows GL-Cache to achieve a $228\times$ higher throughput on average. Moreover, GL-Cache achieves a slight improvement in hit ratio compared to LRB, with a 7% increase on average and 25% at P90 (10% of the traces) compared to LRB. Compared to the learned cache with the highest hit ratio, GL-Cache increases the hit ratio by 3% on average and 13% at the P90 tail, with a 64% higher throughput. Varying group sizes allow GL-Cache to change learning granularity, leading to a spectrum of algorithms. Along with two other system parameters, this spectrum enables users to navigate the trade-off between efficiency and throughput.

This chapter makes the following contributions.

- It classifies existing learned caches into three categories based on learning granularity and propose a new approach for learning in caching — group-level learning. group-level learning amortizes overheads over objects in the group to achieve high throughput. By leveraging multiple group features and accumulating more training signals, group-level learning also achieves a high hit ratio.
- It presents the design and implementation of GL-Cache, which overcomes the challenges by using group-level learning to achieve high cache efficiency with low-overhead learning. For the first time, a group-level utility function is defined and used for cache eviction.
- GL-Cache was evaluated using a diverse set of 118 production traces to illustrate and understand the high efficiency and high throughput of group-level learning.

Chapter 7

Cache Eviction with Lazy Promotion and Quick Demotion

Compared to existing learned caches, GL-Cache achieves a low overhead with a low miss ratio. However, learning still poses a large overhead compared to simple heuristics. Moreover, we find that while block cache workloads can achieve high efficiency using GL-Cache, many web cache workloads do not benefit from group-level learning. When comparing GL-Cache with Segcache, which uses the simple FIFO-merge eviction algorithm, I found that Segcache often achieves a lower miss ratio on the Twitter key-value cache workloads. I dug deeper to understand the reason, which led to the discovery of the two new techniques that will be introduced in this chapter.

7.1 Introduction

Caching is a well-known and widely deployed technique to speedup data accesses [45, 87, 91, 114, 121, 166, 167, 170, 211, 281, 308, 383], reduce repeated computation [125, 290, 370] and data transfer [42, 43, 64, 202, 312, 317, 320, 373]. The core of a cache is the eviction algorithm, which chooses the objects stored in the limited cache space. Two metrics describe the performance of an eviction algorithm: efficiency measured by the miss ratio and throughput measured by the number of requests served per second.

The study of cache eviction algorithms has a long history [39, 90, 92, 310] with a majority of the work centered around LRU. LRU maintains a doubly-linked list, *promoting* objects to the head of the list upon cache hits and evicting the object at the tail of the list when needed. Belady and others found that memory access patterns often exhibit temporal locality — "the most recently used pages were most likely to be reused in the immediate future". Thus, LRU using *recency* to promote objects was found to be better than FIFO [39, 93].

Most eviction algorithms designed to achieve high efficiency start from LRU. For example, many algorithms such as ARC [227], SLRU [177, 250], 2Q [172, 177, 206], MQ [397] and multi-generational LRU [244], use multiple LRU queues to separate hot and cold objects. Some algorithms, e.g., LIRS [165] and LIRS2 [393], maintain an LRU queue but use different metrics to promote objects. While other algorithms, e.g., LRFU [105], EE-LRU [309], LeCaR [329] and CACHEUS [291], augment LRU's recency with different metrics. In addition, many recent works, e.g., Talus [36], improve LRU's ability to handle scan and loop requests.

Besides efficiency, there have been fruitful studies on enhancing LRU's throughput performance and thread scalability. Each cache hit in LRU promotes an object to the head of the queue, which requires updating at least six pointers guarded by locks. These overheads are not acceptable in many deployments that need high performance [32, 115, 276, 354]. Thus, performance-centric systems often use FIFO-based algorithms to reduce LRU's overheads. For example, FIFO-Reinsertion and CLOCK variants [90, 258, 310] have been developed as LRU approximations. It is often perceived that these algorithms trade miss ratio for better throughput and scalability [33, 118, 166, 172, 258].

Via a large-scale simulation study on 6587 traces with 858 billion requests from 12 sources collected in the past two decades, we make a case for breaking away from LRU completely and instead designing eviction algorithms based on FIFO.

FIFO provides many benefits compared to LRU, including less metadata, less computation, better scalability [115, 372] and flash friendliness [70, 226, 340, 378]. However, FIFO alone often leaves a large efficiency headroom. To reduce FIFO's miss ratio while retaining its benefits, we introduce two techniques — LAZY PROMOTION and QUICK DEMOTION.

LAZY PROMOTION (LP) performs promotion only at the eviction time. An example technique is "reinsertion", which puts the eviction candidate back into the cache if requested since the last insertion. Common wisdom suggests that FIFO with LAZY PROMOTION is a form of weak LRU and less efficient than LRU [33, 118, 166, 172, 258]. However, *our large-scale simulation study shows that such "weak LRUs" are more efficient than*

LRU (section 7.3).

Quick Demotion (QD) removes *most* objects quickly after they are inserted. We show that the opportunity cost of waiting for new objects to traverse through the queue(s) is too high. We demonstrate the importance of QD by modifying five state-of-the-art eviction algorithms (including non-LRU based) — adding a small probationary FIFO queue (10% of the cache size) and a metadata-only ghost queue to track FIFO eviction history. Evaluations show that QD-enhanced algorithms reduce the miss ratio from the corresponding state-of-the-art algorithm by 2.8% on average on the 6587 traces. In comparison, ARC can only reduce LRU's miss ratio by 6.4%, and QD-enhanced ARC reduces LRU's miss ratio by 7.9%.

We further demonstrate a simple eviction algorithm QD-LP-FIFO by applying the aforementioned Lazy Promotion and Quick Demotion on top of FIFO. QD-LP-FIFO uses a probationary FIFO queue (QD) in front of a main cache using the CLOCK eviction algorithm (FIFO + LP). QD-LP-FIFO is simple yet efficient. Our evaluations on the 6587 traces show that QD-LP-FIFO achieves lower miss ratios than state-of-the-art eviction algorithms. For example, QD-LP-FIFO reduces LIRS's miss ratios by 1.8% and LeCaR's miss ratios by 4.9% on average. We believe that further innovations in better Lazy Promotion and Quick Demotion techniques will lead to a class of simple and efficient eviction algorithms. Moreover, we envision that future eviction algorithms can be designed like building LEGO by adding different LP and QD techniques to a base algorithm such as FIFO.

This chapter makes two main contributions:

- Contrary to the common belief that LRU approximations are less efficient, we show that FIFO with Lazy Promotion (e.g., FIFO-Reinsertion/CLOCK) achieves a lower miss ratio than LRU.
- We demonstrate that Quick Demotion is critical for efficient caching. Equipped with QD, a simple FIFO-based algorithm can have a lower miss ratio than complex state-of-the-art designs.

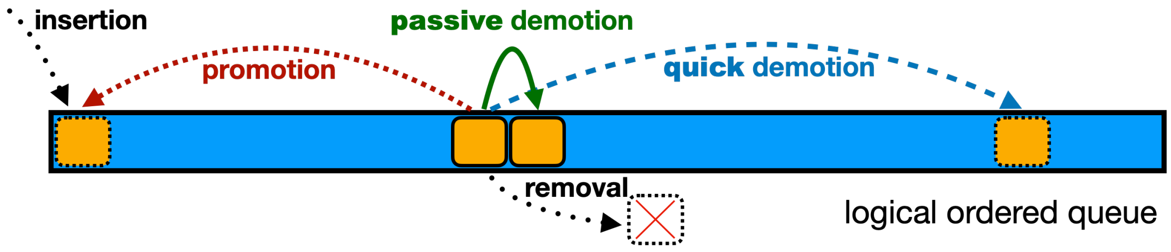


Figure 7.1: The cache abstraction.

7.2 Why FIFO and What it needs

The benefits of FIFO over LRU have been explored in many previous works [115, 118, 370, 372]. For example, FIFO has less metadata (if any) and requires no metadata update on each cache hit, and thus is faster and more scalable than LRU. In contrast, LRU requires updating six pointers on each cache hit, which is not friendly for modern computer architecture due to random memory accesses and extensive locking. Moreover, FIFO is always the first choice when implementing a flash cache because it does not incur write amplification [41, 70, 226, 378]. Although FIFO has throughput and scalability benefits, it is well-known that FIFO provides lower efficiency (higher miss ratio) than LRU.

To understand the various factors that affect the miss ratio, we introduce a cache abstraction. A cache can be viewed as a logically total-ordered queue with four operations: `insertion`, `removal`, `promotion`, and `demotion`. Objects in the cache can be compared and ordered based on some metric (e.g., time since the last request), and the eviction algorithm evicts the least valuable object based on the metric. `Insertion` and `removal` are user-controlled operations, where `removal` can either be directly invoked by the user or indirectly via the use of time-to-live (TTL). `Promotion` and `demotion` are cache internal operations used to maintain the logical ordering between objects.

We observe that most eviction algorithms use `promotion` to update the ordering between objects. For example, all the LRU-based algorithms promote objects to the head of the queue on cache hits, which we call `eager promotion`. Meanwhile, `demotion` is performed implicitly: when an object is promoted, other objects are passively demoted. We call this process `passive demotion`, a slow process as objects need to traverse through the cache queue before being evicted. However, we will show that instead of eager promotion and passive demotion, eviction algorithms should use Lazy Promotion

Table 7.1: Datasets used in this work (traces with less than 1 million requests or 10,000 objects are excluded).

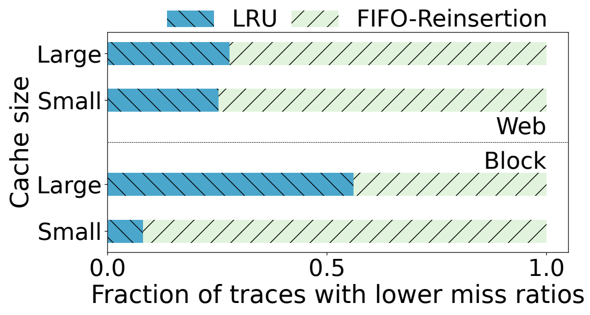
trace collections	approx time	# traces	cache type	# request (million)	# object (million)
MSR [252]	2007	13	block	410	74
FIU [182]	2008	9	block	514	20
Cloudphysics[333]	2015	106	block	2,114	492
CDN 1	2018	219	object	3,728	298
Tencent Photo [394]	2018	2	object	5,650	1,038
Wiki CDN [349]	2019	3	object	2,863	56
Tencent CBS [387, 389]	2020	4030	block	33,690	551
Alibaba [2, 198, 341]	2020	652	block	19,676	1702
Twitter [370]	2020	54	KV	195,441	10,650
Social Network 1	2020	219	KV	549,784	42,898
CDN 2	2021	1273	object	37,460	2,652
Social Network 2	2021	7	mixed	6,807	1,436

(section 7.3) and Quick Demotion (section 7.4).

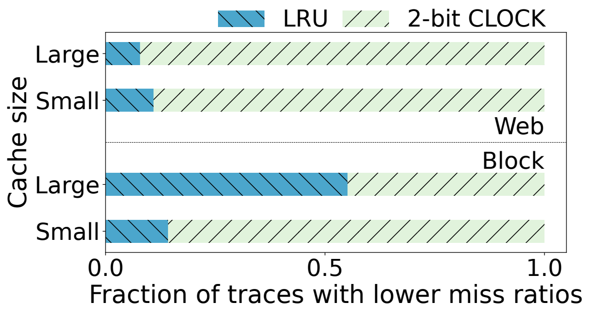
7.3 Lazy Promotion

To avoid popular objects from being evicted in FIFO while not incurring much performance overhead, we propose adding Lazy Promotion on top of FIFO (called LP-FIFO), which *promotes objects only when they are about to be evicted*. Lazy Promotion aims to retain popular objects with minimal effort. An example of LP-FIFO is FIFO-Reinsertion: an object is inserted back at eviction time if it has been requested while in the cache.

LP-FIFO has several benefits over always promotion (promoting on every access) used in LRU-based algorithms. First, LP-FIFO inherits FIFO's throughput and scalability benefits because few metadata operations are needed when an object is requested. For example, FIFO-Reinsertion only needs to update a Boolean field upon the *first* request to a cached object without locking. Second, performing promotion at eviction time allows the cache to make better decisions by accumulating more information about the objects, e.g., how many times an object has been requested.



(a) FIFO-Reinsertion (CLOCK) vs LRU



(b) 2-bit CLOCK vs LRU

Figure 7.2: The fraction of the 6587 traces on which an algorithm has a lower miss ratio. FIFO-Reinsertion and 2-bit CLOCK are more efficient than LRU, with a lower miss ratio on most traces.

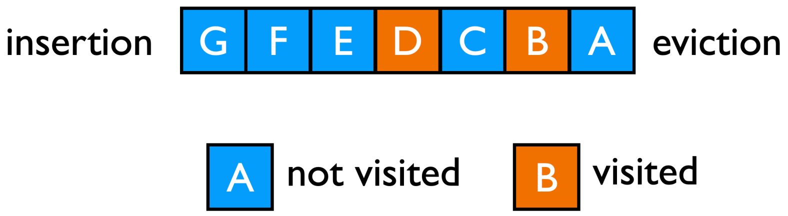


Figure 7.3: LAZY PROMOTION often leads to QUICK DEMOTION. Using FIFO-Reinsertion as an example, the newly-inserted object G will be pushed down by both objects requested before (e.g., B, D) and after G. In contrast, only objects requested after G can push G down in LRU.

To understand LP-FIFO's efficiency, we performed a large-scale simulation study on 6587 production traces from 12 data sources (Table 7.1), which include open-source and proprietary datasets collected between 2007 and 2021. The 12 datasets contain 858 billion (11,311 TB) requests and 61.8 billion (2,114 TB) objects, and cover different types of caches, including block, KV, and object caches. We further divide the traces into block and web (including Memcached and CDN). We choose small/large cache size as 0.1%/10% of the number of unique objects in the trace.

We compare the miss ratios of LRU with two LP-FIFO algorithms: FIFO-Reinsertion [1] and 2-bit CLOCK. 2-bit CLOCK tracks object frequency up to three, and an object's frequency decreases by one each time the CLOCK hand scans through it. Objects with

[1]Note that FIFO-Reinsertion, 1-bit CLOCK, and Second Chance are different implementations of the same eviction algorithm.

zero frequency are evicted.

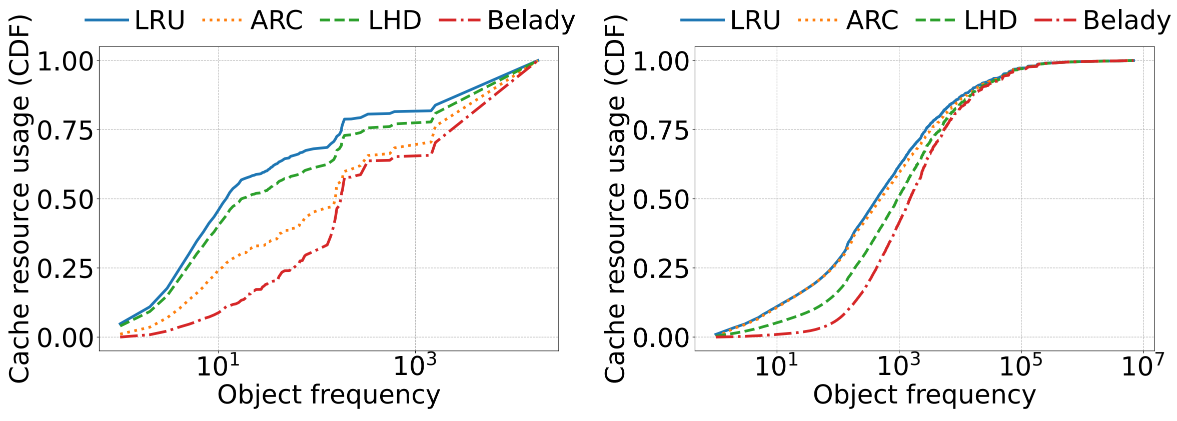
Common wisdom suggests that these two LP-FIFO examples are LRU approximations and will exhibit higher miss ratios than LRU [33, 118, 166, 172, 258]. We suspect this impression came from the 1960s when LRU and CLOCK were designed for virtual memory page replacement. We conjecture that CLOCK may not work as well as LRU for such workloads because LRU can better adapt to sudden working set changes. According to Denning, memory access patterns show abrupt changes between phases [93]. However, we do not observe such patterns in the block and web cache workloads. However, we found that **LP-FIFO often exhibits miss ratios lower than LRU**. Figure 7.2 shows that FIFO-Reinsertion is better than LRU on more than 70% of the web traces across the two cache sizes and over 90% of the block traces on the small cache size. Similarly, over 90% of the web traces favor 2-bit CLOCK over LRU, and this observation holds across datasets. In addition, FIFO-Reinsertion and 2-bit CLOCK reduce LRU's miss ratio by 1% and 2.5% on average (not shown in the figure) across all the traces.

Two reasons contribute to LP-FIFO's high efficiency. First, Lazy Promotion often leads to Quick Demotion (section 7.4). For example, under LRU, a newly-inserted object G is pushed down the queue only by 1) new objects and 2) cached objects that are requested after G. However, besides the objects requested after G, the objects requested before G (but have not been promoted, e.g., B, D) also push G down the queue when using FIFO-Reinsertion (Figure 7.3). Second, compared to promotion at each request, object ordering in LP-FIFO is closer to the insertion order, which we conjecture is better suited for many workloads that exhibit popularity decay — old objects have a lower probability of getting a request.

While LP-FIFO surprisingly wins over LRU in miss ratio, it cannot outperform state-of-the-art algorithms. We next discuss another building block that bridges this gap.

7.4 Quick Demotion

Efficient eviction algorithms not only need to keep popular objects in the cache but also need to evict unpopular objects fast. In this section, we show that Quick Demotion (QD) is critical for an efficient eviction algorithm, and it allows FIFO-based algorithms to achieve state-of-the-art efficiency.



(**a**) MSR trace (hm0) (**b**) Twitter trace (cluster52)

Figure 7.4: Cache resource consumption by objects in different algorithms. More efficient algorithms spend fewer resources on unpopular objects.

Because demotion happens passively in most eviction algorithms, an object typically traverses through the cache before being evicted. Such traversal gives each object a good chance to prove its value to be kept in the cache. However, cache workloads often follow Zipf popularity distribution [25, 41, 51, 370] with most objects being unpopular. This is further exacerbated by 1) the scan and loop access patterns in the block cache workloads [38, 291, 329], and 2) the vast existence of dynamic and short-lived data, the use of versioning in object names, and the use of short TTLs in the web cache workloads [370]. We believe the *opportunity cost of new objects demonstrating their values is often too high*: the object being evicted at the tail of the queue may be more valuable than the objects recently inserted.

Removing low-value objects faster is not a new idea and has been discussed under various contexts, such as removing scan pages [38, 165], correlated accesses [172], and one-hit wonders [219, 348]. These observations have inspired eviction algorithms such as 2Q [172], MQ [397], ARC [227], SLRU [154], LHD [38], and Hyperbolic [48]. However, we find that the demotion in existing algorithms is often not fast enough.

We compute and plot how different algorithms spend cache resources on objects of varying popularity. The resource consumption of an object is calculated using $C_{obj} = \sum(T_{eviction} - T_{insertion}) \times S_{obj}$ similar to the idea in the previous work [38]. Throughout this work, we assume objects to be uniform in size so that we can focus on the effect of access patterns on efficiency. Figure 7.4 shows two representative traces, and Table 7.2 shows

Table 7.2: The miss ratios of the algorithms in Figure 7.4.

algorithm/workload	LRU	ARC	LHD	Belady
MSR	0.5263	0.4899	0.5131	0.4438
Twitter	0.2005	0.1841	0.1756	0.1309

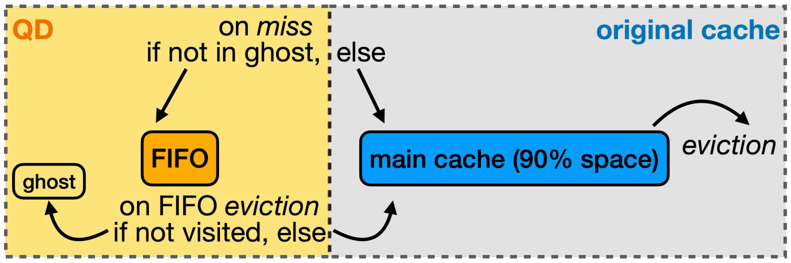
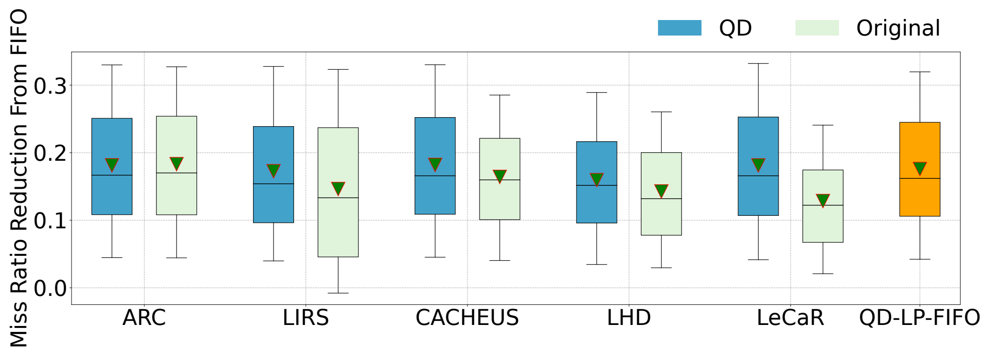


Figure 7.5: An example of QD: add a probationary FIFO queue to an existing cache.

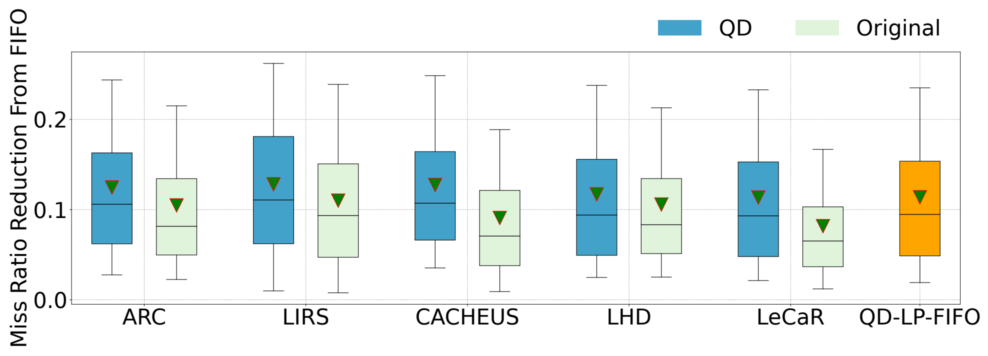
the corresponding miss ratios. ARC and LHD often spend fewer resources on unpopular objects than LRU, thus showing lower miss ratios. Between ARC and LHD, ARC spends fewer resources on unpopular objects and has a notably lower miss ratio than LHD on the MSR trace. We have a similar observation on the Twitter trace as well. Moreover, among all four algorithms, Belady [39] always spends the fewest resources on unpopular objects and has significantly lower miss ratios. In summary, efficient algorithms often spend fewer resources on unpopular objects.

To further illustrate the importance of QUICK DEMOTION, we mount a simple QD technique on top of state-of-the-art eviction algorithms (Figure 7.5). The QD technique consists of a small probationary FIFO queue storing cached data and a ghost FIFO queue storing metadata of objects evicted from the probationary FIFO queue. The probationary FIFO queue uses 10% of the cache space and acts as a filter for unpopular objects: objects not requested after insertion are evicted early from the FIFO queue. The main cache runs a state-of-the-art algorithm and uses 90% of the space. And the ghost FIFO stores as many entries as the main cache. Upon a cache miss, the object is written into the probationary FIFO queue unless it is in the ghost FIFO queue, in which case, it is written into the main cache. When the probationary FIFO queue is full, if the object to evict has been accessed since insertion, it is inserted into the main cache. Otherwise, it is evicted and recorded in the ghost FIFO queue.

We add this FIFO-based QD technique to five state-of-the-art eviction algorithms,



(**a**) Block workloads, small size (0.1% of all objects)



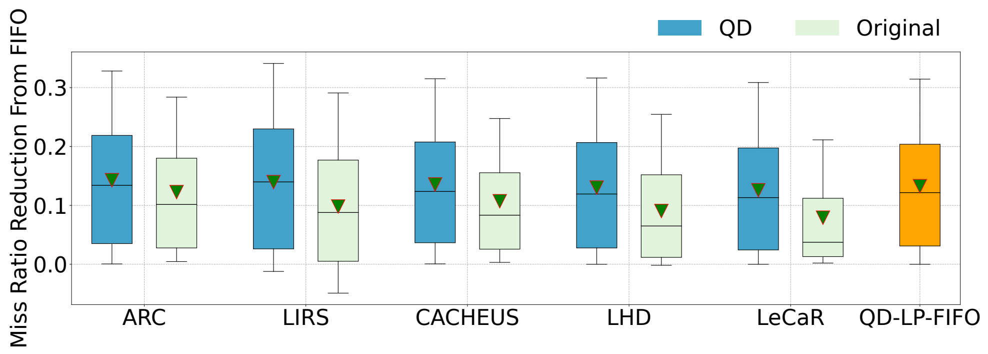
(**b**) Block workloads, large size (10% of all objects)

Figure 7.6: Evaluated on the block cache traces, QD-enhanced algorithms outperform state-of-the-art algorithms at both small and large cache sizes. QD-LP-FIFO achieves similar or better miss ratio reduction compared to state-of-the-art algorithms.

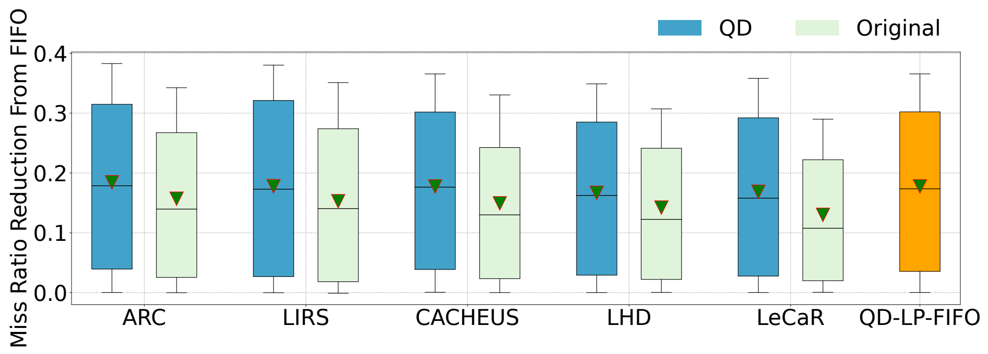
ARC [227], LIRS [165], CACHEUS [291], LeCaR [329] and LHD [38]. We used the open-source LHD implementation from the authors, implemented the others following the corresponding papers, and cross-checked with open-source implementations [2]. We evaluated the QD-enhanced and original algorithms on the 6587 traces. Because the traces have a wide range of miss ratios, we choose to present each algorithm's miss ratio reduction from FIFO calculated as $\frac{mr_{FIFO} - mr_{algo}}{mr_{FIFO}}$.

Figure 7.6 and Figure 7.7 show that the QD-enhanced algorithms further reduce the miss ratio of each state-of-the-art algorithm on almost all percentiles. For example, QD-ARC (QD-enhanced ARC) reduces ARC's miss ratio by up to 59.8% with a mean reduction of 1.5% across all workloads on the two cache sizes, QD-LIRS reduces LIRS's

[2]All state-of-the-art algorithms are complex, and we found two different open-source LIRS implementations used in previous works have bugs.



(**a**) Web workloads, small size (0.1% of all objects)



(**b**) Web workloads, large size (10% of all objects)

Figure 7.7: Evaluated on the web cache traces, QD-enhanced algorithms outperform state-of-the-art algorithms at both small and large cache sizes. QD-LP-FIFO achieves similar or better miss ratio reduction compared to state-of-the-art algorithms.

miss ratio by up to 49.6% with a mean of 2.7%, and QD-LeCaR reduces LeCaR's miss ratio by up to 58.8% with a mean of 4.8%. Note that because of the large number and wide coverage of traces, the best state-of-the-art algorithm ARC can only reduce the miss ratio by 14.2% and 6.4% on average compared to FIFO and LRU.

The gap between the QD-enhanced algorithm and the original algorithm is wider 1) when the state-of-the-art is relatively weak, 2) when the cache size is large, and 3) on the web workloads. With a weaker state-of-the-art, the opportunity for improvement is larger, allowing QD to provide more prominent benefits. For example, QD-LeCaR reduces LeCaR's miss ratios by 4.8% average, larger than the reductions on other state-of-the-art algorithms. When the cache size is large, unpopular objects spend more time in the cache, and Quick Demotion becomes more valuable. For example, QD-ARC and ARC have similar miss ratios on the block workloads at the small cache size. But QD-ARC reduces

ARC's miss ratio by 2.2% on average at the large cache size. However, when the cache size is too large, e.g., 80% of the number of objects in the trace (not shown), we observe that adding QD may increase the miss ratio. At last, QD provides more benefits on the web workloads than the block workloads. We conjecture that web workloads have more short-lived data and exhibit stronger popularity decay, which leads to a more urgent need for QUICK DEMOTION.

While QUICK DEMOTION can reduce miss ratios, it further increases the complexity of the already complicated state-of-the-art algorithms. To reduce complexity, we add the same QD technique on top of the 2-bit CLOCK discussed in section 7.3 and call it QD-LP-FIFO. QD-LP-FIFO uses two FIFO queues to cache data and a ghost FIFO queue to track evicted objects. It is not hard to see QD-LP-FIFO is simpler than all state-of-the-art algorithms — it requires at most one metadata update on a cache hit and no locking for any cache operation. Therefore, we believe it will be faster and more scalable than all state-of-the-art algorithms. Besides enjoying all the benefits of simplicity, QD-LP-FIFO also achieves lower miss ratios than state-of-the-art algorithms (Figure 7.6 and Figure 7.7). For example, compared to LIRS and CACHEUS, QD-LP-FIFO reduces miss ratio by 1.8% and 2.3% on average across the 6587 traces. While the goal of this work is not to propose a new eviction algorithm, QD-LP-FIFO illustrates how we can build simple yet efficient eviction algorithms by adding QUICK DEMOTION and LAZY PROMOTION techniques to a simple base eviction algorithm such as FIFO.

7.5 Discussions

LP and QD techniques. We have demonstrated reinsertion as an example of LP (section 7.3) and the use of a small probationary FIFO queue as an example of QD (section 7.4). However, these are not the only techniques. For example, reinsertion can leverage different metrics to decide whether the object should be reinserted. Besides reinsertion, several other techniques are often used to reduce promotion and improve scalability, e.g., periodic promotion [276], batched promotion [345], promoting old objects only [41], promoting with try-lock [148]. Although these techniques do not fall into our strict definition of LAZY PROMOTION (promotion on eviction), many of them effectively retain popular objects from being evicted. On the QUICK DEMOTION side, besides the small probationary FIFO queue, one can leverage other techniques to define and discover unpopular objects such as Hyperbolic [48] and LHD [38]. Moreover, admission algorithms, e.g., TinyLFU [108, 109, 110], Bloom Filter [43, 219], probabilistic [41] and ML-based [113] admission algorithms, can be viewed as a form of QD — albeit some of them are too aggressive at demotion (rejecting objects from entering the cache).

We remark that QD bears similarity with some generational garbage collection algorithms [94, 275] which separately store short-lived and long-lived data in young-gen and old-gen heaps. Therefore, ideas from garbage collection may be borrowed to strengthen cache eviction algorithms.

We believe that the design of QD-LP-FIFO opens a door to designing simple yet efficient cache eviction algorithms by innovating on LP and QD techniques. And we envision future eviction algorithms can be designed like building LEGO — adding LAZY PROMOTION and QUICK DEMOTION on top of a base eviction algorithm.

Why "X" is not better than QD-LP-FIFO. Eviction algorithms that use multiple queues (e.g., ARC, 2Q, and 2Q variants in many production systems [41, 206, 228, 250]) share similarities with QD-LP-FIFO. However, there are two major differences between QD-LP-FIFO and previous works. First, QD-LP-FIFO only uses FIFO queues, and promotion to a different queue (e.g., main cache) only happens when an object is being evicted. Second, QD-LP-FIFO uses a *tiny* fixed-size FIFO queue (10% of cache size) for QUICK DEMOTION, while previous works use *much larger* (e.g., 50% of cache size) or adaptive queue sizes. Ideally, the adaptive algorithms (e.g., ARC) should provide similar or lower miss ratios than QUICK DEMOTION. However, our study suggests otherwise. There are a few reasons behind this. First, the adaptive algorithms' methods to adjust queue size are not optimal. For ARC, we observe that manually limiting the queue size and slowing down the queue size adjustment often reduce miss ratios. Second, LAZY PROMOTION is resistant to request bursts and better suited for workloads with popularity decay (section 7.3). We observe that replacing the LRU queues in ARC with FIFO-Reinsertion also reduces the miss ratio. In general, adaptive algorithms, such as ARC and CACHEUS, adapt their parameters based on a limited number of past requests, which may not predict the future well.

Limitations. Throughout this work, to focus on how access patterns affect cache efficiency, we ignore other factors, such as object size and TTL, which are important for web cache workloads. While the LAZY PROMOTION and QUICK DEMOTION techniques we have discussed are not size-aware, designing size-aware LAZY PROMOTION and QUICK DEMOTION techniques are worth pursuing in the future.

7.6 Chapter Summary

To the best of our knowledge, this is by far the largest eviction algorithm study — 60,000× larger than previous work [291] in terms of request count. Contrary to the common belief, we discover that LP-FIFO (e.g., FIFO-Reinsertion/CLOCK) is better than LRU in terms of miss ratio (in addition to its well-known benefits on throughput and scalability). Moreover, we demonstrate the importance of QUICK DEMOTION for efficient caching by adding a probationary FIFO queue to five state-of-the-art eviction algorithms. The QD-enhanced algorithms can further improve the state-of-the-art algorithms' efficiency. This study illustrates the importance of LAZY PROMOTION and QUICK DEMOTION for eviction algorithms' efficiency. And it demonstrates a new LEGO-like approach to designing future eviction algorithms.

Chapter 8

S3-FIFO: FIFO Queues are All You Need

The previous chapter demonstrates an efficient cache eviction algorithm that needs LAZY PROMOTION and quick demotion. Moreover, it shows that adding a small FIFO queue on top of state-of-the-art algorithms to filter out most new and unpopular objects can improve their efficiency.

This chapter will examine **why** QUICK DEMOTION is critical for an efficient cache eviction algorithm. This chapter will show that most of the objects in the cache are one-hit wonders. Therefore, quickly removing them is critical for cache efficiency. Moreover, while many complex techniques are used in state-of-the-art eviction algorithms, surprisingly, the effective component is just evicting new objects very quickly. With this discovery, this chapter will demonstrate a new algorithm that combines a small FIFO queue and a large FIFO-Reinsertion queue. The small FIFO queue is used for QUICK DEMOTION, while the reinsertion in the large FIFO queue achieves LAZY PROMOTION. This new eviction algorithm, S3-FIFO, is not only efficient but also very scalable because FIFO queues can be implemented using lock-free queues.

8.1 Background

Software caches are ubiquitously deployed today, e.g., inside end-user devices [179, 202], at the edge of the Internet [26, 34, 43, 57, 117, 119, 248, 261, 300, 317, 364, 365, 373], and across system stacks in a data center [106, 115, 121, 224, 259, 267, 281, 290, 311, 355, 383, 384]. While the data stored in different types of caches have different names, e.g., block, page, object, and asset, we use the term "objects" for ease of discussion.

8.1.1 Metrics of a cache

The heart of a cache is the eviction algorithm, which decides the objects to store in the limited space.

Efficiency. A more *efficient* (sometimes called more "effective") eviction algorithm retains more useful objects in the cache and provides a lower *miss ratio*, which measures the fraction of requests that must be fetched from the backend. While request miss ratio is the most common efficiency metric, some cache deployments aiming to reduce bandwidth usage, e.g., proxy caches, also evaluate *byte miss ratio*: the fraction of bytes that need to be fetched from the origin.

Throughput. A cache's throughput measures the number of requests it can serve per second (QPS). Having higher throughput reduces the number of CPU cores required to serve a workload.

Scalability. Modern CPUs have a large number of cores. For example, AMD EPYC 9654P has 192 cores [11]. A cache's scalability measures how its throughput increases with the number of CPU cores. Ideally, a cache's throughput would scale linearly with the number of CPU cores. However, in many eviction algorithms, read operations necessitate metadata updates under locking. Therefore, they cannot fully harness the computation power of modern CPUs.

Flash writes. While DRAM is the most commonly used storage medium for caching, many systems today also use flash for its higher density, lower price, and lower power consumption. Flash lifetime becomes a critical metric when using flash for caching because flash only supports a limited number of writes [7, 49, 225, 320]. Moreover, small random writes on flash cause device-level write amplification, which not only reduces the flash lifetime but also increases read and write tail latency [143, 144, 196, 367]. To achieve a more manageable flash lifetime, most production flash cache systems, e.g., Apache Trafficserver [14], Memcached Extstore [230], Cachelib large object cache [41], and Google Colossus flash cache [378], use FIFO or FIFO-reinsertion. Besides the flash eviction algorithm, many systems also employ admission algorithms, e.g., bloom filter or machine-learning-based algorithms, to select "good" data to write to flash [69, 113].

Simplicity and generality. A cache eviction algorithm's complexity and generality are two additional factors that play a critical role in its adoption. While complexity is often inversely correlated with throughput and scalability, a simple design can offer benefits beyond just improved performance metrics, such as fewer bugs and reduced maintenance overhead. Linux Kernel developers stated that "Predicting which pages will be accessed in the near future is a tricky task. The kernel not only often gets it wrong, but it also wastes a lot of CPU time to make the incorrect choice" [205]. Generality is crucial for similar reasons. If the same data structure and eviction algorithm can be used for different types of caches, it can help reduce the development and maintenance overheads. A similar argument can also be found in previous work from Meta [41].

8.1.2 Prevalence of LRU-based cache

Cache workloads exhibit temporal locality: recently accessed data are more likely to be re-accessed. Therefore, Least-Recently-Used (LRU) is more efficient than FIFO and is widely used in DRAM caches [41, 50, 228, 330]. Moreover, advanced eviction algorithms designed to improve efficiency are mostly built upon LRU. For example, ARC [227], SLRU [177], 2Q [172], EELRU [309], LIRS [165], TinyLFU [109], LeCaR [329], and CACHEUS [291] all use one or more LRU queues to order objects.

Albeit efficient, LRU and LRU-based algorithms have three problems. First, LRU is often implemented using a doubly-linked list, requiring two pointers per object, which becomes a large overhead when the object is small. As a result, Twitter and Meta have designed specialized compact caches for workloads having small objects [41, 97, 372].

Second, LRU promotes objects to the head of the queue (called promotion) upon each cache hit, which performs at least six random memory accesses protected by a lock, significantly limiting the cache's scalability [121, 268]. For example, the RocksDB developers "confess" that the LRU caches in RocksDB are the scalability bottleneck [104]. Therefore, a new cache using CLOCK [90] eviction has been implemented to address this problem in 2022 [289].

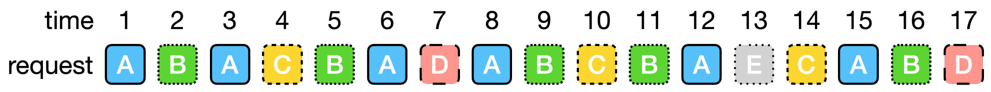
Third, LRU is not flash-friendly. The object eviction order in LRU is different from the insertion order, which leads to random writes on flash, and reduces flash lifetime.

8.2 Motivation

While the last few decades of eviction algorithm study are centered around LRU, we believe modern eviction algorithms should be designed with FIFO queues instead of LRU queues. FIFO can be implemented using a ring buffer without per-object pointer metadata, and it does not promote an object upon each cache hit, thus removing the scalability bottleneck. However, FIFO falls behind LRU and state-of-the-art eviction algorithms in efficiency.

What does FIFO need? The primary limitation of FIFO is its inability to retain frequently accessed objects, so the most straightforward improvement is to insert these objects back. FIFO-Reinsertion [1] is an algorithm that keeps track of object access and reinserts accessed objects during eviction. Compared to LRU, FIFO-Reinsertion incurs a lower overhead on a cache hit, requiring no operation or just an atomic set for the first request to an object. However, reinsertion alone is insufficient, and FIFO-Reinsertion still lags behind state-of-the-art eviction algorithms on efficiency (subsection 8.4.2).

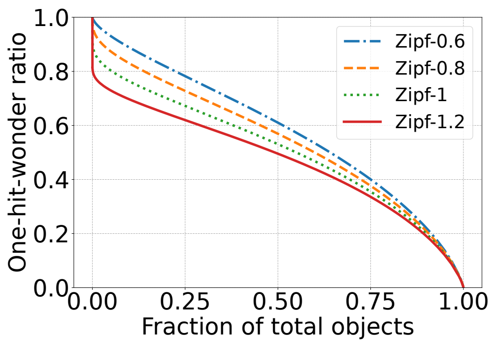
Our insight is that a cache experiences more one-hit wonders (objects having no access after insertion) than what common full trace analyses suggested [219, 348], highlighting the importance of swiftly removing most new objects. Specifically, we observe a median one-hit-wonder ratio of 26% across 6594 production traces. However, for a random request sequence containing 10% of unique objects in the trace, 72% of the objects have only one request in the sequence.



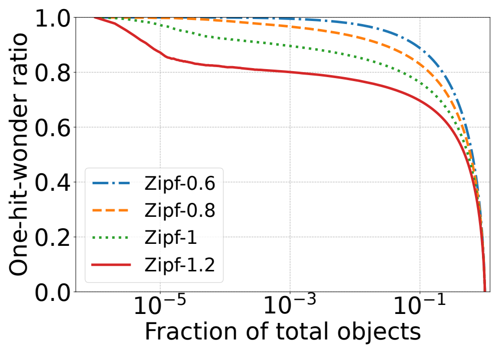
start time	end time	sequence length (# objects)	# one-hit wonder	one-hit-wonder ratio
1	17	5	1 (E)	20%
1	7	4	2 (C, D)	50%
1	4	3	2 (B, C)	67%

Figure 8.1: A shorter sequence has a higher one-hit-wonder ratio.

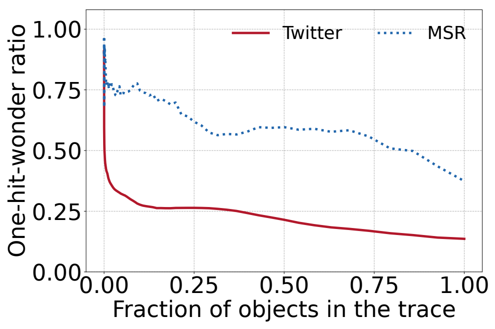
[1]FIFO-Reinsertion, Second chance, and CLOCK are different implementations of the same algorithm.



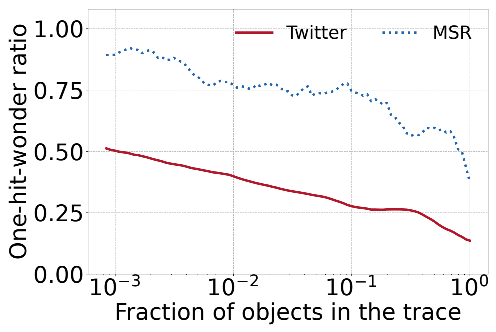
(**a**) Zipf trace, linear-scale



(**b**) Zipf trace, log-scale



(**c**) Production traces, linear-scale



(**d**) Production traces, log-scale

Figure 8.2: Left two: the one-hit-wonder ratio decreases with sequence length (as a fraction of the unique objects in the full sequence) for synthetic Zipf traces. Different curves show different skewness α. We plot both linear and log-scale X-axis for ease of reading. Right two: production traces show similar observations. Note that the X-axis shows the fraction of objects in the trace, much smaller than the number of possible objects in the backend. Therefore, the production curves capture the left region of the Zipf curves.

8.2.1 More one-hit wonders than expected

The term "one-hit-wonder ratio" measures the fraction of objects that are requested only once in a trace. It is commonly used in content delivery networks (CDNs) due to large one-hit-wonder ratios [34, 219].

Although the one-hit-wonder ratio varies between different types of cache workloads, we find that shorter request sequences (consisting of fewer unique objects) often have higher one-hit-wonder ratios. In the subsequent analysis, *we measure sequence length using the number of unique objects*.

Figure 8.1 illustrates this observation using a toy example. The request sequence comprises seventeen requests for five objects, out of which one object (E) is accessed once. Thus, the one-hit-wonder ratio for the sequence is 20%. Considering a shorter sequence from the 1^{st} to the 7^{th} request, two (C, D) of the four unique objects are requested only once, which leads to a one-hit-wonder ratio of 50%. Similarly, the one-hit-wonder ratio of a shorter sequence from the 1^{st} to 4^{th} request is 67%. More formally, we make the following observation.

Observation. *Assume that the object popularity of a request sequence follows the Zipf distribution with the least popular object having one request, and there are M unique objects in total. Then the one-hit-wonder ratio of the complete sequence is $\frac{1}{M}$. For any sub-sequence ending with a one-hit wonder, if the sub-sequence contains C unique objects, the expected one-hit-wonder ratio $\mathcal{F}(x = C)$ monotonically decreases with the sequence length x measured in the number of objects.*

The intuition is that most objects are unpopular (rank higher than $C + 1$ in Zipf distribution for a cache of size C) and have an expected number of requests between 0 and 1. If they show up in the sub-sequence, it is very likely that they will not get another request within the sub-sequence.

This setting can be viewed as a variant of the coupon-collector problem where we have M unique coupons in total, and the probability of collecting coupon i follows the Zipf distribution. We would like to know the number of coupons we have collected only once when we have C unique coupons.

We use Monte Carlo simulations to find how $\mathcal{F}(x)$ changes with the sequence length x (measured in the number of objects). We first generate Zipf request traces of different skewness α under independent reference model [74], then take random sub-sequences and measure the one-hit-wonder ratios. We repeat 100 times and report the mean. The results are plotted in Figure 8.2a and Figure 8.2b. We show both linear and log-scale X axes for clarity. The one-hit-wonder ratio decreases with increasing sequence length. Between different curves, more skewed workloads exhibit lower one-hit-wonder ratios at the same sequence length because unpopular objects have a lower probability of appearing in more skewed workloads.

We have also performed the same measurement on production traces. Figure 8.2c and Figure 8.2d show a block trace (MSR hm_0) and a web trace from Twitter (cluster 52). The curves look different from the Zipf curves at first glance. This is because the production traces are not long enough to capture all objects in the backend systems, and it is not possible to know the total number of objects that can be requested. As a result, the X-axis

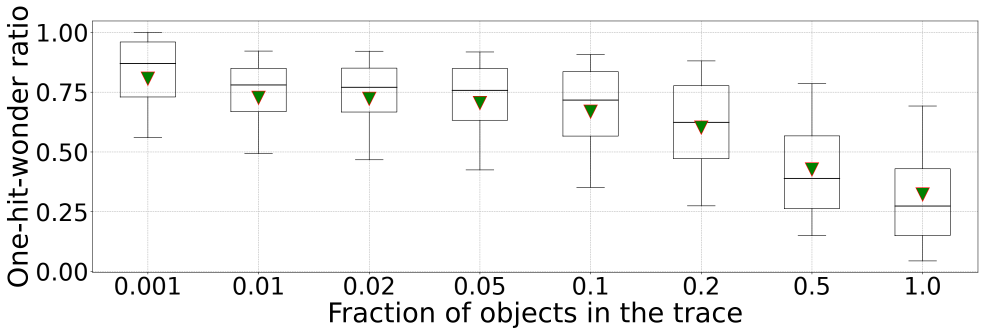


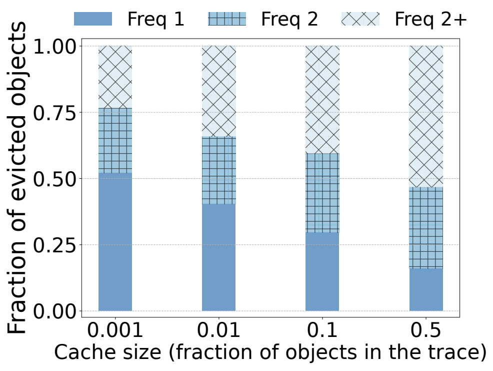
Figure 8.3: The one-hit-wonder ratio across 6594 traces (Table 8.1). The whiskers show P10 and P90, and the triangle shows the mean.

shows the fraction of objects in the trace. Therefore, the production curves only capture the left region of the synthetic curves, and we observe that they match the synthetic curves. For example, when comparing Figure 8.2a and Figure 8.2c, we see curves in both figures have steep drops at the beginning before slowing down. Moreover, the Twitter trace is known to be more skewed [370], and it shows a larger drop than the MSR trace, which matches the observation on the Zipf traces. Compared to the one-hit-wonder ratio of the full trace at 13% (Twitter) and 38% (MSR), a random sub-sequence containing 10% objects shows a one-hit-wonder ratio of 26% on the Twitter trace and 75% on the MSR trace. The increase is more significant when the sequence length is further reduced.

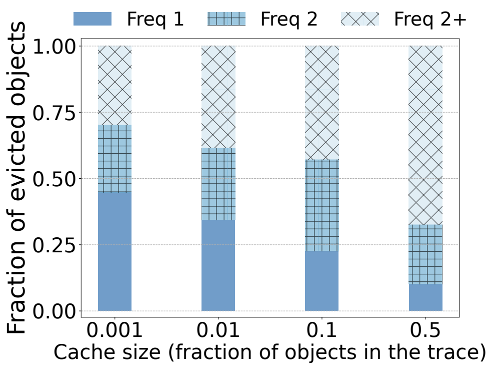
We further evaluated 6594 production traces (more details in Table 8.1). Figure 8.3 shows the one-hit-wonder ratios of all traces at different sequence lengths. Compared to the full traces with a median one-hit-wonder ratio of 26%, sequences containing 50% of the objects in the trace show a median one-hit-wonder ratio of 38%. Moreover, sequences with 10% and 1% of the objects exhibit one-hit-wonder ratios of 72% and 78%, respectively.

Because the cache size is always much smaller than the trace footprint (the number of objects in the trace), evictions start after encountering a short sequence of requests. This observation suggests that if the cache size is set as 10% or 1% of the trace footprint, approximately 72% and 78% of the objects would not be reused before eviction.

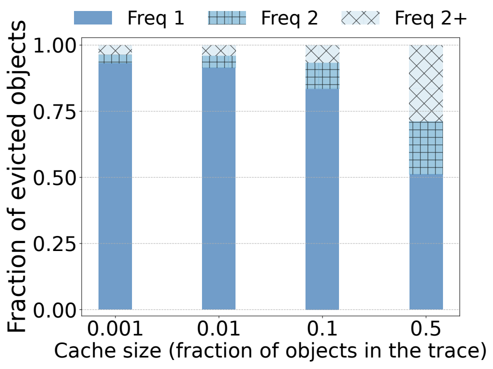
We further corroborate the observation with cache simulations. Figure 8.4 shows the distribution of object frequency at eviction. Our trace analysis (Figure 8.2d) shows that the Twitter trace has a 26% one-hit-wonder ratio for sequences of 10% trace length. The simulation shows a similar result: 26% and 24% of the objects evicted by LRU and Belady are not requested after insertion at a cache size of 10% of the trace footprint. Similarly,



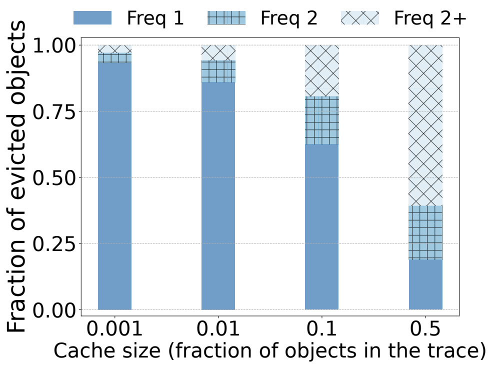
(**a**) Twitter trace, LRU



(**b**) Twitter trace, Belady



(**c**) MSR trace, LRU



(**d**) MSR trace, Belady

Figure 8.4: The frequency of objects at eviction.

the MSR trace exhibits a higher one-hit-wonder ratio of 75% for sequences of 10% trace length (Figure 8.2d), and Figure 8.4 shows that 82% and 68% of the objects evicted by LRU and Belady have no reuse. *This suggests that these one-hit wonders are often good eviction candidates, and one may not need highly sophisticated eviction algorithms*.

8.2.2 The need for QUICK DEMOTION

Based on the observation, a cache should filter out these one-hit wonders because they occupy space without providing benefits. It is a common practice to employ Bloom Filters to reject one-hit wonders from entering the cache in CDNs [219, 348]. However, a Bloom Filter rejects objects too fast with a lack of precision since it rejects all objects that have not been seen before. It causes the second requests to all objects to be cache misses, which often leads to mediocre efficiency (subsection 8.4.2).

Filtering out one-hit wonders bears some resemblance to designing scan-resistant cache eviction algorithms, as objects requested during a scan are often one-hit wonders. Researchers have developed a variety of algorithms for storage workloads that can avoid cache pollution and thrashing caused by scanning requests, e.g., ARC [227], LRU-K [263], 2Q [172], EELRU [309], LIRS [165], LeCaR [329], CACHEUS [291], and LHD [38]. However, existing algorithms cannot guarantee the minimum and maximum time one-hit wonders stay in the cache before being removed. We find these algorithms sometimes evict too fast or too slowly, and their complexities make it difficult to reason about the behavior (subsection 8.5.1).

This raises the question: can we simply use a small probationary FIFO queue to guarantee that one-hit wonders are removed after a fixed number of objects are inserted?

8.3 Design and implementation

As mentioned in subsection 8.1.1, a cache eviction algorithm needs to be simple and scalable besides being efficient. This section presents S3-FIFO, a **s**imple and **s**calable eviction algorithm that consists of only **s**tatic FIFO queues.

We start by defining the LRU queue and FIFO queue. An *LRU queue* updates object ordering during cache hits by promoting the requested object to the head of the queue. A *FIFO queue* does not update ordering during cache hits, and objects are evicted in the insertion order. However, evicted objects may be reinserted into the queue to preserve hot objects. As mentioned in subsection 8.1.2, most eviction algorithms are built with LRU queue, and only a few algorithms, e.g., FIFO-Reinsertion, use FIFO queue because conventional wisdom suggests LRU queue can provide a lower miss ratio.

8.3.1 S3-FIFO design

S3-FIFO uses three FIFO queues: a small FIFO queue ($\mathcal{S}$), a main FIFO queue ($\mathcal{M}$), and a ghost FIFO queue ($\mathcal{G}$). We choose $\mathcal{S}$ to use 10% of the cache space based on experiments with 10 traces and find that 10% generalizes well. $\mathcal{M}$ then uses 90% of the cache space. The ghost queue $\mathcal{G}$ stores the same number of ghost entries (no data) as $\mathcal{M}$.

Cache read. S3-FIFO uses two bits per object to track object access status [375] similar to

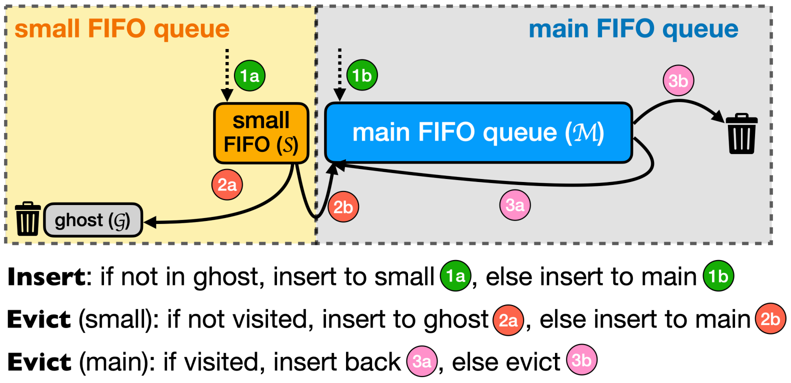

small FIFO queue
main FIFO queue
1a
1b
3b
small FIFO (S)
main FIFO queue (M)
2a
2b
3a
ghost (G)
Insert: if not in ghost, insert to small 1a, else insert to main 1b
Evict (small): if not visited, insert to ghost 2a, else insert to main 2b
Evict (main): if visited, insert back 3a, else evict 3b

Figure 8.5: An illustration of S3-FIFO.

a capped counter with frequency up to 3. Cache hits in S3-FIFO atomically increment the counter by one. Note that most requests for popular objects require no update.

Cache write. New objects are inserted into $\mathcal{S}$ if not in $\mathcal{G}$. Otherwise, it is inserted into $\mathcal{M}$. When $\mathcal{S}$ is full, the object at the tail is either moved to $\mathcal{M}$ if it is accessed more than once or $\mathcal{G}$ if not. And its access bits are cleared during the move. When $\mathcal{G}$ is full, it evicts objects in FIFO order. $\mathcal{M}$ uses an algorithm similar to FIFO-Reinsertion but tracks access information using two bits. Objects that have been accessed at least once are reinserted with one bit set to 0 (similar to decreasing frequency by 1). We illustrate the algorithm in Figure 8.5 and the pseudo-code below.

Algorithm 1 Read in S3-FIFO algorithm.

```
Require: The requested object $x$, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$
1: function READ($x$)
2:     if $x$ in $S$ or $x$ in $M$ then                          ▷ Cache Hit
3:         $x$.freq ← min($x$.freq + 1, 3)            ▷ Frequency is capped to 3
4:     else                                                        ▷ Cache Miss
5:         insert($x$)
6:         $x$.freq ← 0
7:     end if
8: end function
```

Handling different access patterns. One important pattern we identified in subsection 8.2.1 is the large one-hit-wonder ratio a cache experiences due to the limited cache space. The small FIFO queue $\mathcal{S}$ can quickly evict these one-hit wonders so they do not occupy the cache for a long time. This allows S3-FIFO to save the precious cache space for

Algorithm 2 Insert in S3-FIFO algorithm.

```
Require: The requested object $x$, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$, ghost FIFO queue $\mathcal{G}$
 1: function INSERT($x$)
 2:     while cache is full do
 3:         evict()
 4:     end while
 5:     if $x$ in $\mathcal{G}$ then
 6:         insert $x$ to head of $\mathcal{M}$
 7:     else
 8:         insert $x$ to head of $\mathcal{S}$
 9:     end if
10: end function
```

Algorithm 3 Evict in S3-FIFO algorithm.

```
Require: The requested object $x$, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$, ghost FIFO queue $\mathcal{G}$
 1: function EVICT
 2:     if $\mathcal{S}$.size $\geq 0.1 \cdot$ cache size then
 3:         evictS()
 4:     else
 5:         evictM()
 6:     end if
 7: end function
```

Algorithm 4 Evict from the small queue $\mathcal{S}$.

```
Require: The requested object $x$, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$, ghost FIFO queue $\mathcal{G}$
 1: function EVICTS
 2:     evicted $\leftarrow$ false
 3:     while not evicted and $\mathcal{S}$.size $> 0$ do
 4:         $t \leftarrow$ tail of $\mathcal{S}$
 5:         if $t$.freq $> 1$ then
 6:             insert $t$ to $\mathcal{M}$
 7:             if $\mathcal{M}$ is full then
 8:                 evictM()
 9:             end if
10:         else
11:             insert $t$ to $\mathcal{G}$
12:             evicted $\leftarrow$ true
13:         end if
14:         remove $t$ from $\mathcal{S}$
15:     end while
16: end function
```

Algorithm 5 Evict from the main queue $\mathcal{M}$.

```
Require: The requested object $x$, small FIFO queue $\mathcal{S}$, main FIFO queue $\mathcal{M}$, ghost FIFO queue $\mathcal{G}$
 1: function EVICTM
 2:     evicted $\leftarrow$ false
 3:     while not evicted and $\mathcal{M}$.size $> 0$ do
 4:         $t \leftarrow$ tail of $\mathcal{M}$
 5:         if $t$.freq $> 0$ then
 6:             Insert $t$ to head of $\mathcal{M}$
 7:             $t$.freq $\leftarrow$ $t$.freq-1
 8:         else
 9:             remove $t$ from $\mathcal{M}$
10:             evicted $\leftarrow$ true
11:         end if
12:     end while
13: end function
```

more valuable objects. Besides one-hit wonders caused by unpopular objects in skewed cache workloads, many block cache workloads have scan and loop access patterns. Like one-hit wonders, blocks accessed during scans are quickly removed to avoid cache pollution and thrashing. However, blocks not part of a scan but mixed in the scan are also moved to $\mathcal{G}$ in this process. Nevertheless, when these "good" blocks are requested again in the near future, they will be inserted into $\mathcal{M}$ and stay for a longer time.

Table 8.1: Datasets used in this work, the ones with no citation are proprietary datasets. For old datasets, we exclude traces with less than 1 million requests. The trace length used in measuring the one-hit-wonder ratio is measured in the fraction of objects in the trace.

Trace collections	Approx time	Cache type	Duration (days)	# Traces	# Request (million)	Request (TB)	# Object (million)	Object (TB)	One-hit-wonder ratio full trace	10%	1%
MSR [252, 253]	2007	Block	7	13	410	10	74	3	0.56	0.74	0.86
FIU [182]	2008	Block	9-28	9	514	1.7	20	0.057	0.28	0.91	0.91
Cloudphysics [333]	2015	Block	7	106	2,114	82	492	22	0.40	0.71	0.80
CDN 1	2018	Object	7	219	3,728	3640	298	258	0.42	0.58	0.70
TencentPhoto [395]	2018	Object	8	2	5,650	141	1,038	24	0.55	0.66	0.74
WikiMedia CDN	2019	Object	7	3	2,863	200	56	13	0.46	0.60	0.80
Systor [187, 188]	2017	Block	26	6	3,694	88	421	15	0.37	0.80	0.94
Tencent CBS [389]	2020	Block	8	4030	33,690	1091	551	66	0.25	0.73	0.77
Alibaba [198, 341]	2020	Block	30	652	19,676	664	1702	117	0.36	0.68	0.81
Twitter [370]	2020	KV	7	54	195,441	106	10,650	6	0.19	0.32	0.42
Social Network 1	2020	KV	7	219	549,784	392	42,898	9	0.17	0.28	0.37
CDN 2	2021	Object	7	1273	37,460	4,925	2,652	1,581	0.49	0.58	0.64
Meta KV [3]	2022	KV	1	5	1,644	958	82	76	0.51	0.53	0.61
Meta CDN [3]	2023	Object	7	3	231	8,800	76	1,563	0.61	0.76	0.81

8.3.2 Implementation

The FIFO queues can be implemented either using linked lists or ring buffers. Linked-list-based implementation can be added to existing LRU-based caches more easily. However, it has three drawbacks. First, it uses two pointers per object. On workloads with tiny objects [226, 372], this poses a huge storage overhead. Second, traversing through the queue requires random memory accesses. Third, eviction and insertion in linked-list-based implementation require expensive atomic operations: `compare-and-set`, which reduces the scalability.

In contrast, a ring-buffer-based implementation has less overhead and is more scalable but may not be compatible with existing LRU-based caching systems. When using a ring buffer to implement S3-FIFO, the ring buffer maintains the FIFO order, with each slot storing the object or a pointer. Eviction requires bumping the tail pointer in the ring buffer. Although more scalable with lower storage overhead, a ring-buffer-based implementation wastes space when the workload contains many deletion operations because the space of deleted objects cannot be reused until eviction.

Although S3-FIFO has three logical FIFO queues, it can also be implemented with one or two FIFO queue(s). Because objects evicted from $\mathcal{S}$ may enter $\mathcal{M}$, they can be implemented using one queue with a pointer at the 10% mark. However, combining $\mathcal{S}$ and $\mathcal{M}$ reduces scalability because removing objects from the middle of the queue requires locking.

The ghost FIFO queue $\mathcal{G}$ can be implemented as part of the indexing structure. For example, we can store object fingerprint and insertion time of ghost entries in a bucket-based hash table [61, 70, 203, 372]. The fingerprint is a 4-byte hash of the object ID. The insertion time is a virtual timestamp, counting the number of objects inserted into $\mathcal{G}$ thus far. Let $S_{\mathcal{G}}$ denote the size of the ghost queue. If the current time is N (i.e., there were N insertions into $\mathcal{G}$), then all the entries whose timestamp is lower than $N - S_{\mathcal{G}}$ are no longer in $\mathcal{G}$. A ghost entry is removed from the hash table when the object is requested or during hash collision — when the slot is needed to store another entry.

8.3.3 Overhead analysis

Computation. S3-FIFO performs an atomic write upon the first and second request to an object without locking. There is no operation after the second request. Because most requests are for popular objects (more than two requests), S3-FIFO thus performs negligible metadata updates on cache hits. Cache miss requires evicting an object from $\mathcal{S}$ or $\mathcal{M}$. Evicting from $\mathcal{S}$ requires inserting the tail object into $\mathcal{M}$ or $\mathcal{G}$. And evicting from $\mathcal{M}$ may involve reinserting the tail object back to $\mathcal{M}$. However, if an object is not accessed, it requires no reinsertion. Therefore, the number of reinsertions is much smaller than the cache hits in practice. Moreover, removing the tail object and inserting an object to the head of a queue can be implemented lock-free using atomic operations.

Storage. The ghost queue $\mathcal{G}$ stores the same number of objects (without data) as the main queue. Assuming the mean object size is 4 KB, and an object id uses 4 bytes, then $\mathcal{G}$ uses 0.09% of the cache size. Each cached object uses two bits to track access, consuming less than 0.01% of the cache size. Moreover, the two bits can often be piggybacked on unused bits in object metadata. If the FIFO queues are implemented using ring buffers, S3-FIFO can remove the two LRU pointers, saving 16 bytes per object or 0.4% of the cache size.

8.4 Evaluation

In this section, we evaluate S3-FIFO to answer the following questions.

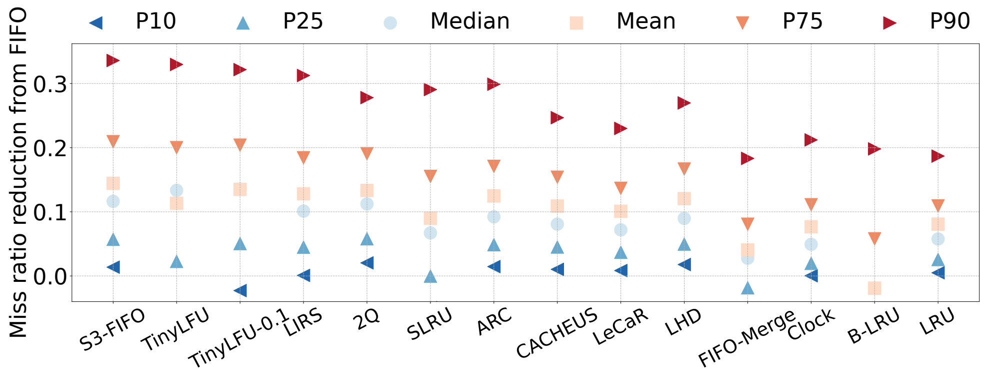
- How does S3-FIFO's efficiency compare with the state-of-the-art eviction algorithms?
- Is S3-FIFO more scalable compared to state-of-the-art?
- Can lessons learned from S3-FIFO help flash cache design?

8.4.1 Evaluation setup

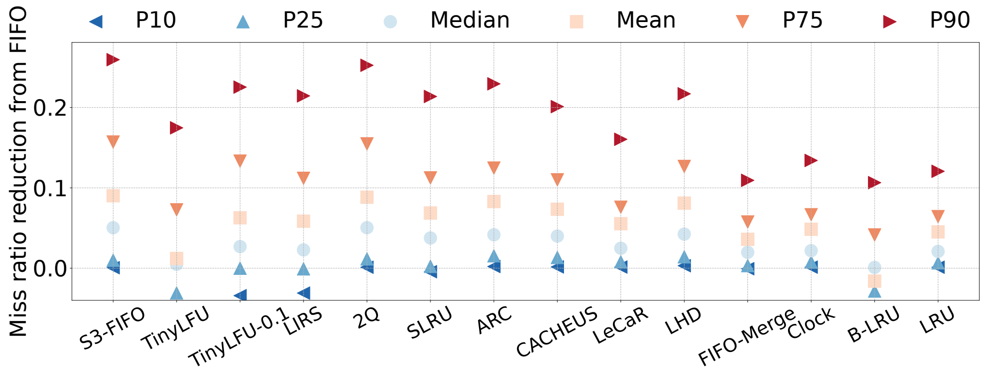
Traces. We evaluated S3-FIFO using a large collection of 6594 production traces from 14 datasets, including 11 open-source and 3 proprietary datasets. These traces span from 2007 to 2023 and cover key-value, block, and object CDN caches. In total, the datasets contain 856 billion requests for 61 billion objects, and 21,088 TB traffic for a total of 3,753 TB of data. Because many large-scale distributed caching systems are multi-tenanted and the traces represent workloads served by more than one server, we split four datasets (CDN 1, CDN 2, Tencent CBS, and Alibaba) with tenant information into per-tenant traces for an in-depth study of the workloads. More details of the datasets can be found in Table 8.1.

Simulator. We implemented S3-FIFO and the state-of-the-art eviction algorithms (described in subsection 8.4.2) in libCacheSim [368]. We referenced and verified the results with multiple open-source simulator implementations [17, 62, 63, 207, 212, 322]. For all state-of-the-art algorithms, we used the parameters described in the original papers. LibCacheSim is designed and tuned for high-throughput cache simulations and can process up to 20 million requests on a single CPU core.

We have also implemented a distributed fault-tolerant computation platform that allows us to run thousands of simulations in parallel. The platform's design does not affect simulation accuracy and is out of the scope of this work. We describe it in a separate



(a) Large cache size, 10% trace footprint



(b) Small cache size, 0.1% trace footprint

Figure 8.6: Each algorithm's miss ratio reduction (from FIFO) at different percentiles across all traces. A larger reduction is better.

blog post [2].

This distributed computation platform and the Cloudlab testbed [107] enable us to evaluate different algorithms and cache sizes on our large datasets (Table 8.1). The simulation processed the datasets in close to 100 passes using different algorithms, cache sizes, and parameters. We estimated that over 80,000 billion requests were processed using a million CPU-core hours.

Unless otherwise mentioned, we ignore object size in the simulator because most production systems use slab storage for memory management, for which evictions are performed within the same slab class (objects of similar sizes). However, we remark that supporting object size is non-trivial for systems that do not use slab-based memory management. Moreover, we do not consider the metadata size in different algorithms,

[2]https://blog.jasony.me/random/tool/2023/08/01/distributed-computation

although S3-FIFO often requires fewer metadata than other algorithms. We evaluated the algorithms at multiple different cache sizes, and we present one large size using 10% of the trace footprint (number of objects in the trace) and one small size at 0.1% of the trace footprint. At 0.1% trace footprint, the cache size may be too small for some traces, so we ignore a trace if the cache size is smaller than 1000 objects. For byte miss ratio evaluation, we considered object size and used the trace footprint in bytes instead of objects.

Because the large number of traces used in the evaluation have a very wide range of miss ratios, we choose to present the miss ratio reduction compared to FIFO: $\frac{MR_{fifo}-MR_{algo}}{MR_{fifo}}$ where MR stands for miss ratio. If an algorithm has a miss ratio higher than FIFO, we calculate FIFO's miss ratio reduction compared to the algorithm and take the negative value: $-\frac{MR_{algo}-MR_{fifo}}{MR_{algo}}$, which bounds the value between -1 and 1. This avoids the impact of outliers on the mean value.

Prototype. We have implemented S3-FIFO in Cachelib [95]. Cachelib uses slab memory management, which pre-allocates all memory during initialization and is highly optimized for LRU-based eviction algorithms. Its extensive usage of metaprogramming and many LRU-based optimizations (e.g., compressed pointers) tightly couple different components. Therefore, we implemented $\mathcal{S}$ and $\mathcal{M}$ using linked lists and $\mathcal{G}$ using a hash table. We implemented a trace replay tool that replays traces in a closed loop for benchmarking. Because the backend often decides the latency and throughput of cache misses, we focus on the cache hit performance and on-demand fill cache misses using pre-generated data object value. We compared S3-FIFO with three algorithms implemented by Cachelib developers: LRU, a variant of 2Q, and TinyLFU. Cachelib developers have devoted huge efforts to improving the throughput and scalability of the three algorithms with techniques such as lock combining, delayed LRU promotion, try-lock-based promotion, and compressed pointers. Besides Cachelib, we also evaluated Segcache, the state-of-the-art scalable key-value cache using open-source code [372].

Open source. We have open-sourced the code and data with more information at the end of the paper.

Evaluation setup. We performed all evaluations on Cloudlab [107]. The simulations used multiple types of nodes from the Clemson site, depending on node availability. The prototype evaluation used c6420 nodes from the Clemson site. We turned off turbo boost, pinned one thread to one core, and used `numactl` to allocate all memory pages on the same NUMA node.

8.4.2 Efficiency (miss ratio)

Miss ratio. The primary criticism of the FIFO-based eviction algorithms is their efficiency, the most important metric for a cache. We compare S3-FIFO with state-of-the-art eviction algorithms designed in the past few decades. The algorithms used in the comparison are either deployed in production or commonly used in other papers. We use all efficiency results from simulation because it allows us to (1) study different types of cache workloads, e.g., block, key-value, and object, (2) focus on and isolate the impact of the eviction algorithm, and (3) requires fewer computation resources to scale up to evaluate the huge datasets. Figure 8.6 shows the (request) miss ratio reduction (compared to FIFO) of different algorithms across traces. At the large cache size, S3-FIFO has the largest reductions across almost all percentiles than other algorithms. For example, S3-FIFO reduces miss ratios by more than 32% on 10% of the traces (P90) with a mean of 14% on the large cache size.

TinyLFU [109] is the closest competitor. TinyLFU uses a 1% LRU window to filter out unpopular objects and stores most objects in a SLRU cache. TinyLFU's good performance corroborates our observation that QUICK DEMOTION is critical for efficiency. However, TinyLFU does not work well for all traces, with miss ratios being lower than FIFO on almost 20% of the traces (the P10 point is below -0.05 and not shown in the figure). This phenomenon is more pronounced when the cache size is small, where TinyLFU is worse than FIFO on close to 50% of the traces.

There are two reasons why TinyLFU falls short. First, the 1% window LRU is too small, evicting objects too fast. Therefore, increasing the window size to 10% of the cache size (TinyLFU-0.1) significantly improves the efficiency at the tail (bottom of the figure). However, increasing the window size reduces its improvement on the best-performing traces (Figure 8.6a). Second, when the cache is full, TinyLFU compares the least recently used entry from the window LRU and main SLRU, then evicts the less frequently used one. This allows TinyLFU to be more adaptive to different workloads. However, if the tail object in the SLRU happens to have a very high frequency, it may lead to the eviction of an excessive number of new and potentially useful objects.

LIRS [165] uses LRU stack (reuse) distance as the metric to choose eviction candidates. Because one-hit wonders do not have reuse distance, LIRS utilizes a 1% queue to hold them. This small queue performs QUICK DEMOTION and is the secret source of LIRS's high efficiency. Similar to TinyLFU, the queue is too small, and it falls short on some cache workloads. However, compared to TinyLFU, fewer traces show higher-than-FIFO miss ratios because the inter-recency metric in LIRS is more robust than the frequency in TinyLFU. In particular, TinyLFU cannot distinguish between many objects with the same low frequency (e.g., 2), but these objects will have different inter-recency values. The downside is that LIRS requires a more complex implementation than TinyLFU.

2Q [172] has the most similar design to S3-FIFO. It uses 25% cache space for a FIFO queue, the rest for an LRU queue, and also has a ghost queue. Besides the difference in queue size and type, objects evicted from the small queue are *not* inserted into the LRU queue. Having a large probationary queue and not moving accessed objects into the LRU queue are the primary reasons why 2Q is not as good as S3-FIFO. Moreover, the LRU queue does not provide observable benefits compared to the FIFO queue (with reinsertion) in S3-FIFO.

SLRU [154, 177] uses four equal-sized LRU queues. Objects are first inserted into the lowest-level LRU queue and promoted to higher-level queues upon cache hits. An inserted object is evicted if not reused in the lowest LRU queue, which performs QUICK DEMOTION and allows SLRU to show good efficiency. However, unlike other schemes, SLRU does not use a ghost queue, making it not scan-tolerant because popular objects mixed in the scan cannot be distinguished. Therefore, we observe that SLRU performs poorly on many block cache workloads (not shown).

ARC uses four LRU queues: two for data and two for ghost entries. The two data queues are used to separate recent and frequent objects. Cache hits on objects in the recency queue promote the objects to the frequency queue. Objects evicted from the two data queues enter the corresponding ghost queue. The sizes of queues are adaptively adjusted based on hits on the ghost queues. When the recency queue is small, newly inserted objects are quickly evicted, enabling ARC's high efficiency. However, ARC is less efficient than S3-FIFO because the adaptive algorithm is not sufficient. We discuss with more details in subsection 8.5.2.

Recent algorithms, including CACHEUS [291], LeCaR [329], LHD [38], and FIFO-Merge [372], are also evaluated. However, we find these algorithms are often less competitive than the traditional ones. In particular, FIFO-merge was designed for log-structured storage and key-value cache workloads without scan resistance. Therefore, similar to SLRU, it performs better on web cache workloads but much worse on block cache workloads.

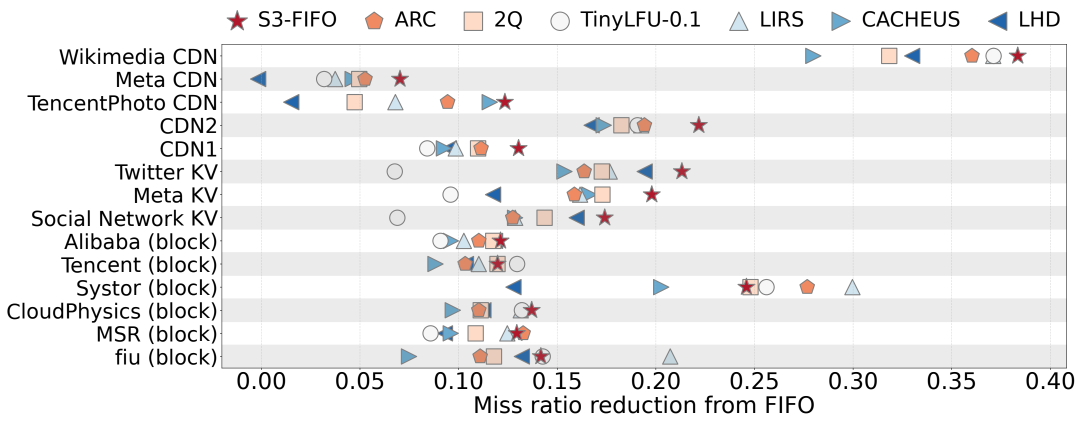
Common algorithms, such as B-LRU (Bloom Filter LRU), CLOCK, and LRU, are weaker than the ones discussed. CLOCK and LRU do not allow QUICK DEMOTION, so their miss ratio reductions are small. B-LRU rejects all one-hit wonders at the cost of the second request for all objects being cache misses. Because of these misses, B-LRU is worse than LRU in most cases. Because an object's second request often arrives *soon* after the first request (temporal locality), the small FIFO queue in S3-FIFO allows these requests to be served as cache hits.

Adversarial workloads for S3-FIFO. We studied the limited number of traces on which S3-FIFO performed poorly and identified one pattern. Most objects in these traces are accessed only twice, and the second request falls out of the small FIFO queue $\mathcal{S}$, which causes the second request to these objects to be cache misses. We remark that these workloads are adversarial for most algorithms that partition the cache space, e.g., TinyLFU, LIRS, 2Q, and CACHEUS. Because the partition for newly inserted objects is smaller than the cache size, it is possible that the second request is a cache hit in LRU and FIFO, but not in these advanced algorithms.

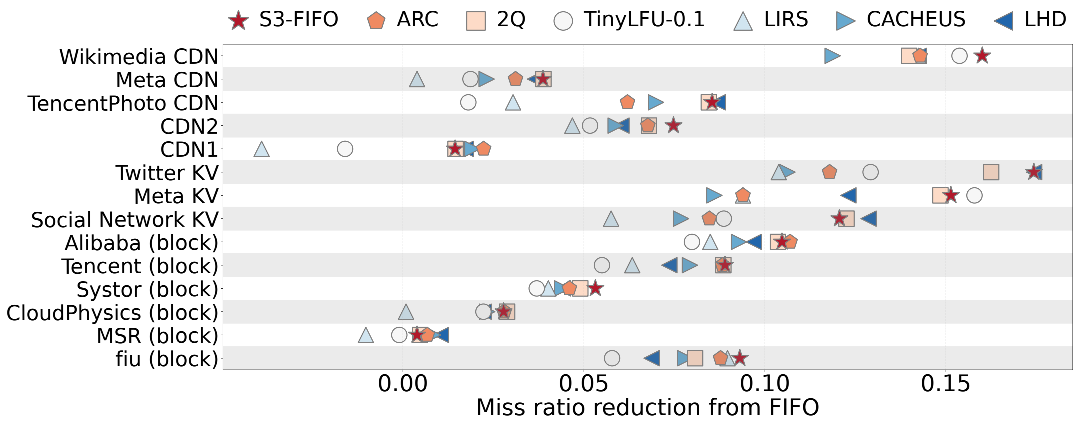
This request pattern resembles a scan because most objects are not requested very soon after the first request. However, it is not a typical scan because any object may show this pattern, and the objects showing this pattern may not be requested consecutively. In our large datasets, we find that the second request often arrives within one minute in these workloads. Therefore, the second request being a miss is a problem only when the cache size is very small, e.g., 1000s of objects. Moreover, using an adaptive algorithm to adjust the queue size can often mitigate the problem, and we discuss more in subsection 8.5.2.

Miss ratio per dataset. We have shown the results across all 6594 traces. However, the number of traces from each dataset differs, and the result could be affected by the dominating dataset. Figure 8.7 shows the mean miss ratio reduction on each dataset using selected algorithms. We observe that S3-FIFO often outperforms all other algorithms by a large margin. Moreover, it is the best algorithm on 10 out of the 14 datasets using a large cache size and 7 out of the datasets using a small cache size. As a comparison, no other algorithm is the best on more than 3 datasets.

Besides being the best on most datasets, S3-FIFO is also more robust than other algorithms — S3-FIFO is among the top three most efficient algorithms on 13 of the 14 datasets at the large cache size. As a comparison, TinyLFU and LIRS are among the top algorithms on



(**a**) Large cache size

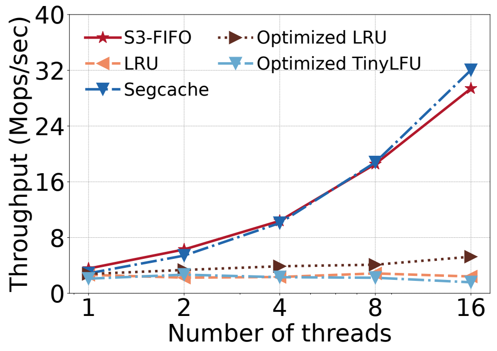


(**b**) Small cache size

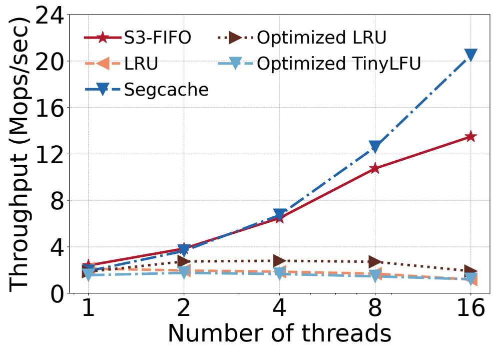
Figure 8.7: The mean miss ratio reduction of different algorithms on each dataset. TinyLFU on the TencentPhoto dataset at the large size is -0.11 and not shown.

some datasets, but on other datasets, they are among the worst algorithms. While it is hard to explain why S3-FIFO is more robust, we conjecture that simplicity contributes to its robustness. In conclusion, we find that QUICK DEMOTION is a key factor for an efficient eviction algorithm. By leveraging this observation, S3-FIFO, a simple algorithm with only FIFO queues, can outperform state-of-the-art.

Byte miss ratio. While (request) miss ratio is important for most cache deployments, CDNs also widely use byte miss ratio to measure bandwidth reduction. We evaluated the same set of eviction algorithms on byte miss ratio. We used the object sizes from each trace and set the cache size to 10% and 0.1% of trace footprint in bytes. The results (not shown due to space limit) are not significantly different from the miss ratio in Figure 8.6.



(**a**) Large cache, LRU miss ratio 0.02



(**b**) Small cache, LRU miss ratio 0.21

Figure 8.8: Throughput scaling with CPU cores on synthetic Zipf $(\alpha = 1.0)$ trace.

Compared to other algorithms, S3-FIFO presents larger byte miss ratio reductions at almost all percentiles. We have also compared S3-FIFO with LRB [312], a machine-learn-based eviction algorithm designed for CDN cache workloads. We used ten random traces (LRB took too long to run on the full dataset), including the Wikimedia traces used in LRB's evaluation. We observe that S3-FIFO and LRB have similar efficiency, although S3-FIFO is much simpler than LRB.

8.4.3 Throughput Performance

S3-FIFO consists of only FIFO queues without locking on either read or write. As a comparison, LRU-based eviction algorithms, such as LRU, 2Q, and TinyLFU, require locking on both cache hits and cache misses. We implemented S3-FIFO in Cachelib to compare the throughput of different algorithms. Because prototype experiments run much longer and cannot be run in parallel, we only evaluated using a synthetic Zipf trace similar to previous work [118]. Moreover, we verified that the miss ratio results from the prototype are consistent with the simulator using a few randomly selected traces. The Zipf workload contains $100 \cdot n_{thread}$ million requests for n_{thread} million 4 KB objects. Figure 8.8 shows that compared to (strict) LRU, the optimized LRU has both higher throughput and better scalability. However, it cannot scale beyond two cores. Compared to LRU, TinyLFU needs to check and update the count-min sketch on cache hits and move objects between the window LRU and the main SLRU on cache misses. Therefore, we observe a lower throughput than LRU due to the extra operations. The optimized 2Q in Cachelib has a similar result (not shown).

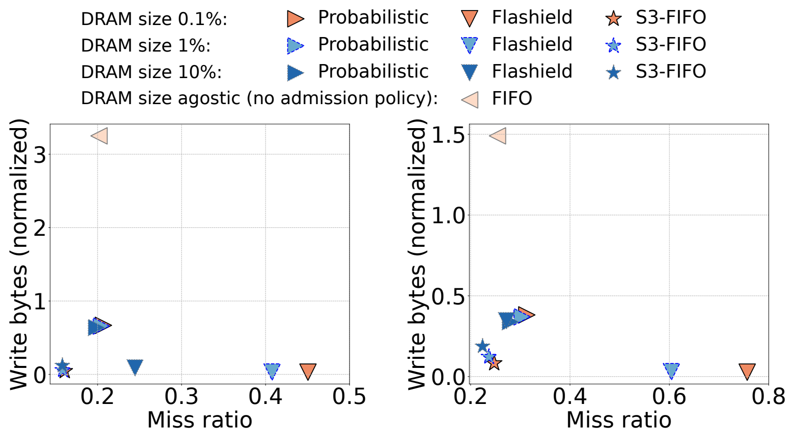


Figure 8.9: The write bytes and miss ratio of no admission control and using different admission algorithms. Both metrics are better when they are lower. Write bytes are normalized to the number of bytes in the trace. Left: Wikimedia CDN trace, right: Tencent Photo CDN trace.

Compared to LRU-based eviction algorithms, S3-FIFO performs fewer operations during cache hits, with a higher throughput on a single thread. Moreover, the lock-free implementation enables the throughput to scale with the number of CPU cores. Under both small and large cache sizes, S3-FIFO runs more than 6× faster than the optimized LRU in Cachelib with 16 threads.

Segcache [372] is the state-of-the-art key-value cache using log-structured storage with the FIFO-Merge eviction algorithm. It uses macro management and FIFO-based eviction to achieve close-to-linear scalability. The macro management enables Segcache to perform much less synchronization — Segcache needs atomic updates only when a segment-chain is changed, which is 100-1000× less frequent than cache misses. However, Segcache is slower than S3-FIFO on a single thread because the merge-based eviction needs to copy data. Moreover, Segcache does not have a comparable efficiency as S3-FIFO as we have shown in Figure 8.6.

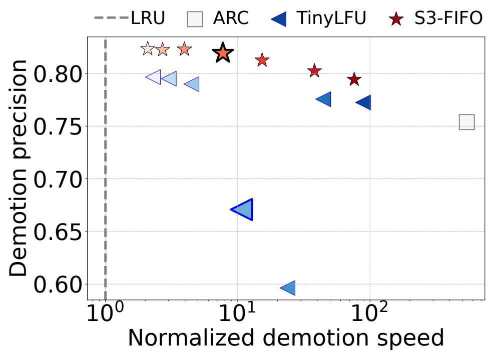
8.4.4 Flash-friendliness

In many flash cache deployments, the flash stores all the cached objects, and DRAM is used for hot objects (and index) [14, 43]. However, writing all data to the flash reduces its lifetime.

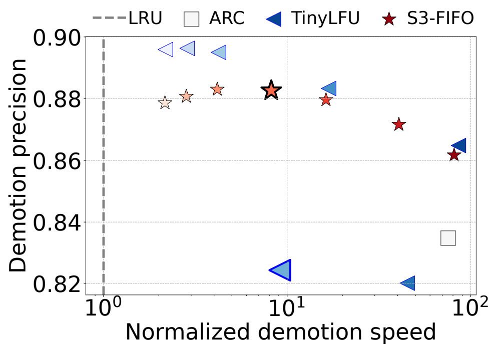
The surprising finding that using a small FIFO queue to perform QUICK DEMOTION can achieve the state-of-the-art miss ratio has an implication for flash cache design. Because most objects evicted from the $\mathcal{S}$ are not worthwhile to be kept in $\mathcal{M}$, we can place $\mathcal{S}$ in DRAM and $\mathcal{M}$ on flash. Objects evicted from DRAM are *not* written to the flash. Only objects requested in $\mathcal{S}$ and $\mathcal{G}$ are written to the flash. This setup reduces *both* flash writes and miss ratio.

Because CDN caches are often deployed using flash, we compare the miss ratio and write bytes using open-source CDN traces from Wikimedia [349] and Tencent Photo CDN [395]. We compare with three schemes. FIFO does not use an admission control and writes everything to the flash. Probabilistic admission uses an LRU DRAM cache and a 20% probability to admit DRAM-evicted objects into the flash cache randomly. Flashield uses a machine learning model (SVM) to predict which objects are worthwhile writing to the flash. S3-FIFO uses a small FIFO and ghost queue in DRAM (0.1%, 1%, 10%) to filter objects, and objects requested at least twice in the DRAM are admitted onto flash. Because the flash cache eviction algorithm is orthogonal to the admission policy, we used FIFO [14, 41, 228] in all experiments (including in S3-FIFO). We have also evaluated other flash-friendly algorithms, such as FIFO-Reinsertion [378], and observed similar results. We set the cache size to 10% of the trace footprint in bytes. We further normalize the write bytes to the number of unique bytes in the trace.

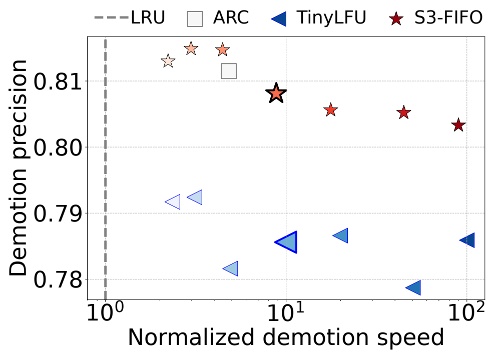
Figure 8.9 shows that compared to no admission control (FIFO), an admission policy can significantly reduce the number of write bytes. However, both probabilistic admission and Flashield trade-off the miss ratio for the reduced write bytes. In contrast, using a small FIFO queue for admission is surprisingly effective at reducing both write bytes and miss ratios. Unlike probabilistic admission, which has almost no dependency on the DRAM size, S3-FIFO and Flashield make admission decisions based on access in DRAM. With a large DRAM (10% of flash cache size), Flashield achieves close to S3-FIFO miss ratio with slightly more writes. However, when the DRAM size is small, objects do not accumulate enough access for the machine-learning model to predict accurately. Meta engineers have also made a similar observation [41].



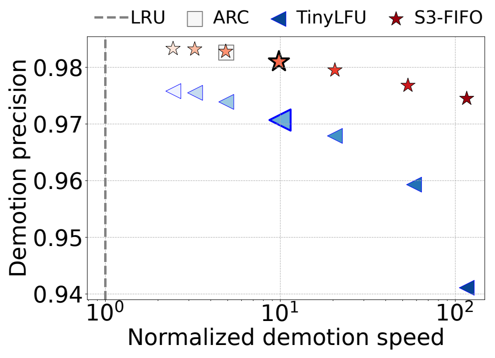
(**a**) Twitter trace, large cache



(**b**) Twitter trace, small cache



(**c**) MSR trace, large cache



(**d**) MSR trace, small cache

Figure 8.10: The normalized mean QUICK DEMOTION speed and precision of different algorithms. TinyLFU and S3-FIFO use different $\mathcal{S}$ sizes (1%, 2%, 5%, 10%, 20%, 30%, and 40% of cache size) and have multiple points with lighter colors representing larger $\mathcal{S}$. The marker of 10% small queue size is highlighted with a larger size.

8.5 Discussion

8.5.1 Why is S3-FIFO effective?

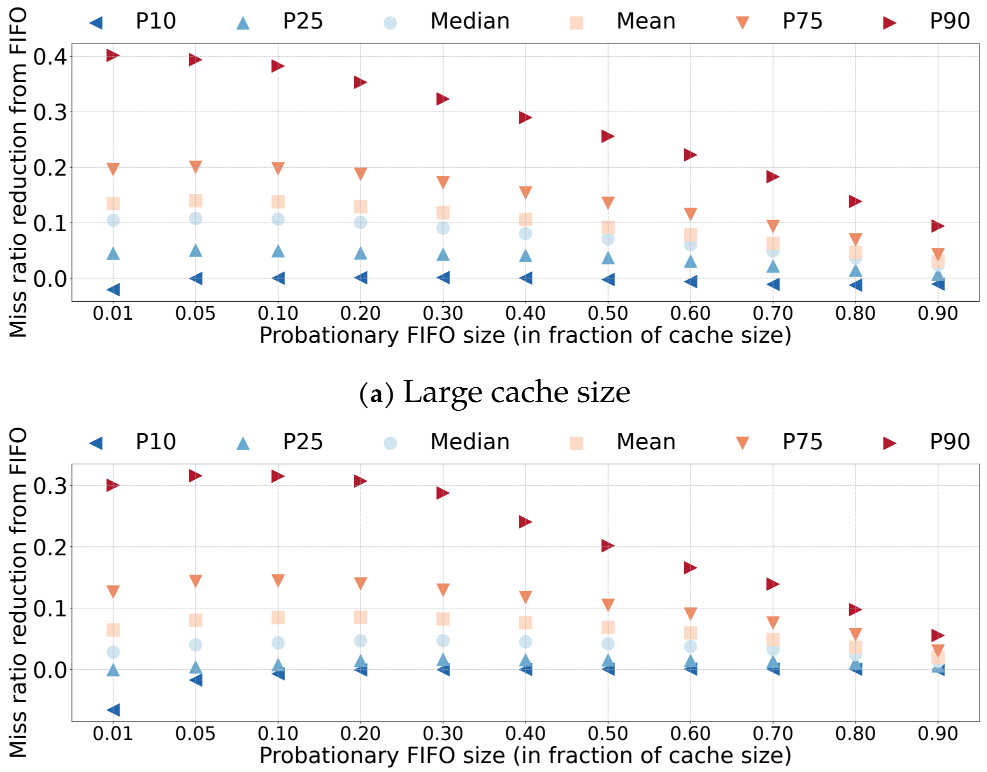
The key to S3-FIFO's efficiency is the small probationary FIFO queue $\mathcal{S}$ that filters out one-hit wonders. Removing low-value items is not new. Admission algorithms, e.g., Bloom Filter, Adaptsize [43], are designed for a similar purpose. However, they reject objects too early and show low efficiency for most cache workloads. Besides admission algorithms, many cache eviction algorithms designed to be scan-resistant, e.g., ARC and 2Q, share a similar idea. They separate new and frequent objects into two queues (denote using $\mathcal{S}$ and $\mathcal{M}$) so that popular objects are not affected by scan requests. This work

Table 8.2: Miss ratio when using different $\mathcal{S}$ sizes (as a fraction of cache size). Increasing $\mathcal{S}$ sizes leads to slower but more accurate QUICK DEMOTION. Thus miss ratio for S3-FIFO first decreases, then increases with $\mathcal{S}$ size. But TinyLFU sometimes shows anomalies. The table should be read together with Figure 8.10. The font color matches the color in Figure 8.10, and the italics show the miss ratio anomaly of TinyLFU.

$\mathcal{S}$ size	0.40	0.30	0.20	0.10	0.05	0.02	0.01
Twitter trace, large cache, ARC miss ratio *0.0483*, LRU miss ratio 0.0488 (Figure 8.10a)							
TinyLFU	0.0451	0.0445	0.0441	*0.0530*	*0.0586*	0.0437	0.0437
S3-FIFO	0.0455	0.0442	0.0432	0.0424	0.0422	0.0422	0.0423
Twitter trace, small cache, ARC miss ratio *0.1941*, LRU miss ratio 0.2005 (Figure 8.10b)							
TinyLFU	0.1744	0.1718	0.1697	*0.1766*	0.1688	*0.1775*	0.1722
S3-FIFO	0.1846	0.1802	0.1765	0.1743	0.1740	0.1752	0.1768
MSR trace, large cache, ARC miss ratio 0.2891, LRU miss ratio 0.3188 (Figure 8.10c)							
TinyLFU	0.2990	0.2949	0.2936	0.2900	0.2893	0.2904	0.2895
S3-FIFO	0.2989	0.2936	0.2896	0.2891	0.2884	0.2887	0.2889
MSR trace, small cache, ARC miss ratio 0.4899, LRU miss ratio 0.5263 (Figure 8.10d)							
TinyLFU	0.4952	0.4923	0.4903	0.4907	0.4922	0.4993	0.5120
S3-FIFO	0.4940	0.4903	0.4890	0.4910	0.4926	0.4953	0.4970

shows that a small static FIFO queue, one of the simplest designs to filter out low-value objects, works better than many more advanced alternatives. But why? We take a closer look at demotion speed and precision using the same trace from section 8.2 to get a deeper understanding. The *normalized* QUICK DEMOTION *speed* measures how long objects stay in $\mathcal{S}$ before they are evicted or moved to $\mathcal{M}$. We use the LRU eviction age as a baseline and calculate the speed as $\frac{\text{LRU eviction age}}{\text{time in } \mathcal{S}}$. We use logical time measured in request count. The QUICK DEMOTION *precision* measures how many objects evicted from $\mathcal{S}$ are not reused soon. Using an idea similar to previous work [312], if the number of requests till an object's next reuse is larger than $\frac{\text{cache size}}{\text{miss ratio}}$, then we say the QUICK DEMOTION results in a correct early eviction.

An algorithm with both faster and more precise QUICK DEMOTION exhibits a lower miss ratio. Figure 8.10 shows that ARC, TinyLFU, and S3-FIFO can quickly demote new objects and have lower miss ratios compared to LRU (Table 8.2).



(**a**) Large cache size

(**b**) Small cache size

Figure 8.11: Miss ratio reduction percentiles using different sizes for the small FIFO. Left: large cache size, right: small cache size.

ARC uses an adaptive algorithm to decide the size of $\mathcal{S}$. We find that the algorithm can identify the correct direction to adjust the size, but the size it finds is often too large or too small. For example, Figure 8.10a shows that ARC chooses a very small $\mathcal{S}$ on the Twitter trace, causing most new objects to be evicted too quickly with low precision. This happens because of two trace properties. First, objects in the Twitter trace often have many requests; Second, new objects are constantly generated. Therefore, objects evicted from $\mathcal{M}$ are requested very soon, causing $\mathcal{S}$ to shrink to a very small size (around 0.01% of cache size). Meanwhile, constantly generated new (and popular) objects in $\mathcal{S}$ face more competition and often have to suffer a miss before being inserted in $\mathcal{M}$, which causes low precision and a high miss ratio (Table 8.2). On the MSR trace, ARC has a reasonable speed with relatively high precision, which correlates with its low miss ratio.

TinyLFU and S3-FIFO have a predictable QUICK DEMOTION speed — reducing the size of $\mathcal{S}$ always increases the demotion speed. When using the same $\mathcal{S}$ size, TinyLFU demotes slightly faster than S3-FIFO because it uses LRU, which keeps some old but recently accessed objects, squeezing the available space for newly inserted objects.

Besides, S3-FIFO often shows higher precision than TinyLFU at a similar QUICK DEMOTION speed, which explains why S3-FIFO has a lower miss ratio. TinyLFU compares the eviction candidates from $\mathcal{S}$ and $\mathcal{M}$, then evicts the less-frequently-used candidate. When the eviction candidate from $\mathcal{M}$ has a high frequency, it causes many worth-to-keep objects from $\mathcal{S}$ to be evicted. This causes not only a low precision but also unpredictable precision and miss ratio cliffs. For example, the precision shows a large dip at 5% and 10% in Figure 8.10a, corresponding to a sudden increase in the miss ratio (Table 8.2).

Although S3-FIFO does not use advanced techniques, it achieves a robust and predictable QUICK DEMOTION speed and precision. As $\mathcal{S}$ size increases, the speed decreases monotonically (moving towards the left in the figure), and the precision also increases until it reaches a peak. When $\mathcal{S}$ is very small, popular objects do not have enough time to accumulate a hit before being evicted, so the precision is low. Increasing $\mathcal{S}$ size leads to higher precision. When $\mathcal{S}$ is very large, many unpopular objects are requested in $\mathcal{S}$ and moved to $\mathcal{M}$, leading to reduced precision as well. Table 8.2 shows that at similar QUICK DEMOTION speed, higher precision always leads to lower miss ratios.

In summary, S3-FIFO *guarantees* that newly inserted unpopular objects are evicted in a predictably short time. The QUICK DEMOTION is often more precise and robust compared to existing approaches. This combination allows S3-FIFO to obtain better than state-of-the-art miss ratios.

8.5.2 How about adaptive eviction algorithms?

Is queue size sensitive? We chose $\mathcal{S}$ to use 10% of the cache size based on results from ten traces and found that it generalizes well across the 6594 traces. Figure 8.11 shows how the miss ratios change with $\mathcal{S}$ size. We observe that a smaller $\mathcal{S}$ leads to larger miss ratio reductions, confirming the importance of QUICK DEMOTION. For example, when the cache size is large, the best-performing traces (P90) have the largest reduction when $\mathcal{S}$ uses 1% of the cache size. However, a smaller $\mathcal{S}$ also causes more traces to have miss ratios higher than FIFO. This aligns with the observation in subsection 8.5.1 where we see smaller $\mathcal{S}$ leads to faster QUICK DEMOTION, but the precision decreases after the peak. Overall, the predictability between efficiency and $\mathcal{S}$ size makes it easy to choose the $\mathcal{S}$ size. And the efficiency does not change much for most traces if $\mathcal{S}$ size is between 5% and 20% of the cache size.

Making queue size adaptive! We designed and implemented an algorithm that adaptively changes the FIFO queue sizes, which we call S3-FIFO-D, S3-FIFO with dynamic queue sizes. S3-FIFO-D maintains a balance between marginal hits on the evicted objects from $\mathcal{S}$ and $\mathcal{M}$. It uses two small ghost queues to track objects evicted from $\mathcal{S}$ and $\mathcal{M}$. Each ghost queue is sized to store 5% of the cached objects (without data). Each time the two ghost queues have more than 100 hits, and one has $2\times$ more hits than the other, S3-FIFO-D moves 0.1% of cache space to the queue whose evicted objects receive more hits. By balancing the marginal hits on the evicted objects, S3-FIFO minimizes the gradient of hits on the evicted objects. If $\mathcal{S}$ is too small, its evicted objects will receive many hits causing an expansion of $\mathcal{S}$. Vice versa. Besides the algorithm described above, we also experimented with another adaptive algorithm similar to ARC, which increases queue size by one upon a hit on the ghost. However, we find this algorithm less robust than S3-FIFO-D.

We compare S3-FIFO-D and S3-FIFO (not shown) and find that S3-FIFO is better than S3-FIFO-D on most traces except the 2% traces at the tail, on which using 10% cache size for $\mathcal{S}$ is far from optimal. In other words, the adaptive algorithm is only useful when the workload is adversarial (which is rare). We tried to tune the parameters in the adaptive algorithm. However, tuning for a few traces is easy, but obtaining good results across traces is very challenging [3].

Where do adaptive algorithms fail? The parameter tuning problem is not unique to S3-FIFO-D. Most, if not all, adaptive algorithms have many parameters. For example, queue resizing requires several parameters, e.g., the frequency of resizing, the amount of space moved each time, the lower bound of queue sizes, and the threshold for trigger resizing. This is not unique for S3-FIFO-D, but also for algorithms such as ARC, whose parameters are less obvious. For example, ARC moves one slot upon a hit on the ghost. But the question remains why one slot instead of half or two? And is it better to handle the hit at the head and tail of the ghost queue differently?

Besides the many hard-to-tune parameters, adaptive algorithms adapt based on observation of the past. However, the past may not predict the future. We find that small

[3]We believe algorithm design should not be tuned on the traces used for evaluation (test dataset), but rather on a validation dataset.

perturbations in the workload often cause the adaptive algorithm to overreact. It is unclear how to balance between under-reaction and overreaction without introducing more parameters. Moreover, some adaptive algorithms, including S3-FIFO-D, implicitly assume that the miss ratio curve is convex because following the gradient direction leads to the global optimum. However, the miss ratio curves of scan-heavy workloads are often not convex [40, 332].

Although we have shown that S3-FIFO is not sensitive to $\mathcal{S}$ size, and the queue size is easier to choose than tuning an adaptive algorithm. We believe adaptations are still important, but how to adapt remains to be explored. For systems that need to find the best parameter, downsized simulations using spatial sampling can be used [332, 333].

8.5.3 LRU or FIFO?

S3-FIFO only uses FIFO queues, but do LRU queues provide better efficiency? We experimented with different queue-type combinations by replacing both the small FIFO queue and the main FIFO queue with LRU queues. And we have also experimented with moving objects from $\mathcal{S}$ to $\mathcal{M}$ upon cache hits and during evictions. Due to space limits, the results are not shown, but we observe that LRU queues *do not* improve efficiency. In particular, using two LRU queues, such as in ARC, is worse than S3-FIFO most of the time. In conclusion, with QUICK DEMOTION, the queue type does not matter.

8.6 Related Work

We have discussed many related works throughout the section 8.1 and subsection 8.4.2. We discuss the rest in this section.

Efficiency-oriented cache design. Besides the eviction algorithms we compared with, many other algorithms are designed to improve the cache efficiency [33, 48, 84, 159, 180, 192, 224, 326, 393]. S3-FIFO differs from existing algorithms in the following way. First, S3-FIFO uses only FIFO queues and does not require promotion on cache hits. Second, S3-FIFO explicitly guarantees the time one-hit wonders stay in the cache before testing popularity. Third, this work shows why a very small probationary cache is needed and uses a smaller probationary queue than most previous works.

Quickly removing one-hit wonders is similar to removing scan/streaming/sequential/looping requests that motivated many previous works [38, 172, 227, 291]. Moreover, similar ideas have also been applied to removing low-priority blocks from lower cache layers in a cache hierarchy [256, 351, 362]. Our previous work also discussed two techniques to improve cache efficiency and scalability — lazy promotion and quick demotion [375]. S3-FIFO is an example of applying the two techniques on FIFO queues to design simple, efficient, and scalable cache eviction algorithms. SIEVE is another eviction algorithm focusing on simplicity, efficiency, and scalability. However, SIEVE is not scan-resistant and only works on web workloads.

Besides eviction algorithms, several other works improved cache efficiency via removing cliffs in miss ratio curves [36], space partitioning [85, 86, 87, 151], prefetching [102, 131, 201, 314, 331, 369], exploiting spatial locality [167], compression [162, 357], leveraging application hints [160, 200, 209, 271, 297, 362], cooperative caching [91, 168, 169, 363], read-write separation [35], reducing metadata, and removing expired objects [372]. These works complement the eviction algorithm designs. However, we remark that integrating multiple designs, e.g., eviction and prefetching are non-trivial and requires additional exploration [28, 56, 65, 131, 314].

Scalability-oriented cache design. Segmented FIFO was designed to achieve low overhead; however, the tradeoff is lower efficiency than LRU [323]. Segcache [372] improves a cache's throughput and scalability by eliminating promotions and locking. Segcache uses log-structured storage to improve scalability. However, Segcache is more efficient for workloads using TTLs. MemC3 [118] and Tricache [121] use CLOCK for scalable data access. However, CLOCK has lower efficiency than S3-FIFO because it cannot quickly remove one-hit wonders. FrozenHot [276] freezes part of the cached data to provide scalable data access and does not improve the eviction algorithm's efficiency.

Besides improving an eviction algorithm, sharding is commonly used to improve scalability. Sharding partitions the key space, and each CPU core serves a slice of the keys. However, cache workloads often follow Zipfian popularity, so sharding leads to load imbalance [117, 150, 155, 211, 281] and limits the whole system's throughput. Besides improving the cache eviction algorithm's scalability, several other works have improved other parts in a key-value cache/store [203]. Compared to these works, S3-FIFO focuses on the eviction algorithm.

Flash endurance. Endurance is a well-known problem for caching on flash. Many works have designed flash-friendly cache eviction algorithms, such as RIPQ [320], SpatialClock [179], and offline algorithms [81]. FlashTier [299], DIDACache [305], Pannier [190] studied the flash cache design beyond eviction algorithms to improve flash cache performance and endurance. Flash cache admission control (also called selective caching in some works) has been explored in LARC [156], WEC [69], S-RAC [257] and SieveStore [274], which use window-based or ghost-based frequency threshold to selectively cache objects on flash. Such designs are similar to using counting Bloom Filter LRU. However, they do not explicitly consider *the role of DRAM to cache new (and unpopular) objects*. This is particularly important as we have shown that B-LRU cannot achieve the optimal efficiency (section 8.4). Flashield [113] and ML-QP [388] track object access in the DRAM cache and use a machine-learning model to decide admission. However, Flashield requires too much DRAM to work. Besides, several works used social features to predict object access patterns [335, 336], which are only applicable in social network cache workloads. While early eviction, selective caching, and selective placement can help with flash endurance, they are also widely used in hierarchical caches to achieve exclusive caching and address the lack of locality. Different algorithms [78, 164, 360, 397], interfaces and systems [130, 351, 361, 362] have been designed to improve the efficiency of hierarchical caches.

8.7 Chapter Summary

This chapter explores the opportunity of building a simple, scalable, yet efficient eviction algorithm with only FIFO queues. The insight is that for any skewed request sequence, the fraction of objects appearing once is much higher in a sub-sequence than in the full trace. Because a cache of size C only observes a short sequence of C objects before evictions, most objects will be one-hit wonders when evicted, even though they may have more requests throughout the full trace. We confirm this observation on 6594 production traces. The median one-hit-wonder ratio of all traces, when considering the entire trace, is 26%. However, when focusing on sequences that comprise 10% of the unique objects in each trace, the median one-hit-wonder ratio skyrockets to 72%.

We leverage this workload property and design S3-FIFO, a **s**imple, **s**calable eviction algorithm with three **s**tatic (fixed-size) FIFO queues. S3-FIFO uses a small probationary FIFO queue to filter out one-hit wonders from entering the main FIFO queue so that cache space can be used for more valuable objects (called *early eviction* or *quick demotion* [375]).

S3-FIFO is not only simple but also efficient. We compare S3-FIFO with 12 eviction algorithms on a large data collection of 6594 production traces from 14 sources. The traces overall contain 856 billion requests collected between 2007 and 2023, and cover block, key-value, and object caches. While advanced algorithms may excel at a few particular workloads, our evaluation shows that S3-FIFO achieves better efficiency (lower miss ratios) across traces at all percentiles than state-of-the-art algorithms. Moreover, S3-FIFO's efficiency is robust. Using a cache size of 10% of objects in the trace, S3-FIFO is the most efficient algorithm on 10 out of the 14 datasets and among the top three most efficient algorithms on 13 datasets. As a comparison, the next best algorithm (LIRS [165]) obtains the highest efficiency on only 2 datasets.

S3-FIFO is also more scalable because FIFO queues enable lock-free implementations. We implemented a prototype in Cachelib and showed that S3-FIFO achieves more than $6\times$ higher throughput than the highly-optimized LRU implementation on 16 cores.

The fact that filtering objects with a small FIFO queue enables better than state-of-the-art efficiency has an implication for flash cache deployments. If the small FIFO queue is in DRAM and the main FIFO queue is on flash, then most objects evicted from DRAM do not need to be written to the flash. This reduces both flash writes and miss ratio. We compare this FIFO filter with a probabilistic filter and a machine-learning-model-based filter from Flashield [113]. The FIFO filter has the lowest miss ratio and the least flash writes evaluated on two open-source CDN traces. Moreover, in contrast to the ML model that requires a large DRAM cache to track object access information for making good decisions, the small FIFO filter excels even when the DRAM cache is only 0.1% of the total cache size.

This chapter makes the following contributions.

- It shows that for cache workloads with skewed popularity, most objects are one-hit wonders at eviction. Therefore, QUICK DEMOTION is critical for cache efficiency.
- Leveraging this new observation, it presents the design and implementation of S3-FIFO, the first FIFO-queue-only eviction algorithm.
- It presents an evaluation comparing S3-FIFO with 12 state-of-the-art eviction algorithms on 6594 traces and shows that S3-FIFO is more efficient and robust.
- It illustrates that S3-FIFO is scalable with $6\times$ higher throughput than an optimized LRU implementation in Cachelib.

Chapter 9

SIEVE: a Simple and yet Efficient Eviction Algorithm

The previous chapter demonstrates how LAZY PROMOTION and QUICK DEMOTION help S3-FIFO achieve high efficiency and close-to-linear scalability. Although S3-FIFO is much simpler than state-of-the-art eviction algorithms, it is still more complex than commonly used simple heuristics, such as LRU. This chapter will illustrate a new eviction algorithm, SIEVE, that further simplifies the design of eviction algorithms. SIEVE uses a *single* queue to achieve both LAZY PROMOTION and QUICK DEMOTION. It is simpler than LRU and achieves state-of-the-art efficiency on web workloads.

9.1 Background and Related Work

9.1.1 Web caches

Web caches are essential components of modern Internet infrastructure, playing a crucial role in reducing data access latency and network bandwidth. Key-value caches, e.g., Memcached [228], Pelikan [97] and Cachelib [95], are widely used in modern web services such as Twitter [370] and Meta [41] to reduce service latency. CDN caches are deployed close to users to reduce data access latency and high WAN bandwidth cost [26, 365, 373, 386].

Cache metrics. Caches are measured along two primary axes: efficiency and throughput performance. Cache efficiency measures how well the cache can store and serve the required data. A cache miss occurs when the requested data is not found in the cache, requiring access to the backend storage to retrieve the data. Common cache efficiency metrics include (1) object miss ratio: the fraction of requests that are cache misses; (2) byte miss ratio: the fraction of bytes that are cache misses. A lower miss ratio indicates higher cache efficiency, as more requests are served directly from the cache, reducing backend load, access latency, and bandwidth costs.

Throughput performance, on the other hand, is as important as efficiency because the goal of a cache is to serve data quickly and help scale the application. Beyond throughput, scalability is also increasingly important [276, 372] as modern CPUs often surpass 100 cores. Scalability measures throughput growth with the number of threads accessing the cache. A more scalable cache can better harness the many cores in a modern CPU.

Access patterns. Web cache workloads typically follow Power-law (generalized Zipfian) distributions [41, 51, 52, 89, 132, 145, 154, 315, 317, 370], where a small subset of objects account for a large proportion of requests. In detail, the i^{th} popular object has a relative frequency of $1/i^{\alpha}$, where α is a parameter that decides the skewness of the workload. Previous works find different α values from 0.6 to 0.8 [51], 0.56 [132], 0.71–0.76 [140], 0.55–0.9 [41], and 0.6–1.5 [370]. The reasons for the large range of α include (1) the different types of workloads, such as web proxy and in-memory key-value cache workloads; (2) the layer of the cache, noting that many proxy/CDN caches are secondary or tertiary cache layers [154]; and (3) the popularity of the service, such as the most popular objects receiving greater volume of requests in more popular (widely-used) web applications. Moreover, web caches often serve constantly growing datasets — new content and objects are created every second.

In contrast, the backend of enterprise storage caches or single-node caches, such as the page cache, often has a fixed size, not regularly observing new objects. Further, many storage cache workloads often have scan and loop patterns [291], in which a range of block addresses are sequentially requested in a short time. Such patterns are rare in web cache workloads according to our observation on 1559 traces from 7 datasets.

9.1.2 Cache eviction policies

The cache eviction algorithm, which decides which objects to store in the limited cache space, governs the performance and efficiency of a cache. The field of cache eviction algorithms has a rich literature [18, 36, 37, 39, 46, 64, 86, 90, 92, 105, 110, 113, 114, 115, 152, 166, 191, 193, 269, 287, 310, 320, 335, 356, 364, 393].

Increasing complexity. Most works on cache eviction algorithms focused on improving efficiency, such as LRU-k [263], TwoQ [172], SLRU [177], GDSF [64], EELRU [309], LRFU [105], LIRS [165], ARC [227], MQ [396], CAR [33], CLOCK-pro [166], TinyLFU [109, 111], LHD [38], LeCaR [329], LRB [312], CACHEUS [291], GLCache [374], and HALP [313]. Over the years, new cache eviction algorithms have gradually convoluted. Algorithms from the 1990s use two or more static LRU queues or use different recency metrics; algorithms from the 2000s employ size-adaptive LRU queues or use more complicated recency/frequency metrics, and algorithms from the 2010s and 2020s start to use machine learning to select eviction candidates. Each decade brought greater complexity to cache eviction algorithms. Nevertheless, as we show in section 9.3, while the new algorithms excel on a few specific traces, they do not show a significant improvement (and some are even worse) compared to the traditional ones on a large number of workloads. The combination of limited improvement and high complexity explains why these algorithms have not been used in production systems.

The trouble with complexity. Multiple problems come with increasing complexity. First, complex cache eviction algorithms are difficult to debug due to their intricate logic. For example, we find two open-source cache simulators used in previous works have two different bugs in the LIRS [165] implementation. Second, complexity may affect efficiency in surprising ways. For example, previous work reports that both LIRS and ARC exhibit Belady's anomaly [137, 332]: miss ratio increases with the cache size for some workloads. It's worth noting that FIFO, although simple, also suffers from this anomaly. Third, complexity often negatively correlates with throughput performance. A more intricate algorithm performs more computation with potentially longer critical sections, reducing both throughput and scalability. Furthermore, many of these algorithms need to store more per-object metadata, which reduces the effective cache size that can be used for caching data. For example, the per-object metadata required by CACHEUS is $3.3\times$ larger than that of LRU. Fourth, complex algorithms often have parameters that can be difficult to tune. For example, all the machine-learning-based algorithms include many parameters about learning. Although some algorithms do not have explicit parameters, e.g., LIRS, previous work shows that the implicit ghost queue size can impact the efficiency [332].

Trade-offs in using simple eviction algorithms. Besides works focusing on improving cache efficiency, several other works have improved cache throughput and scalability. For example, MemC3 [118] uses Cuckoo hashing and CLOCK eviction to improve Memcached's throughput and scalability; MICA [203] uses log-structured storage, data partitioning, and a lossy hash table to improve key-value cache throughput and scalability. Segcache [372] uses segment-structured storage with a FIFO-based eviction algorithm and leverages macro management to improve scalability. Frozenhot [276] improves cache scalability by freezing hot objects in the cache to avoid locking. However, it's crucial to note that while these approaches excel in throughput and scalability, they often compromise on cache efficiency due to the use of simpler, weaker eviction algorithms such as CLOCK and FIFO.

9.1.3 LAZY PROMOTION and QUICK DEMOTION

As discussed in Chapter 7, promotion and demotion are two cache internal operations used to maintain the logical ordering between objects. LAZY PROMOTION and QUICK DEMOTION are two important properties of efficient cache eviction algorithms.

LAZY PROMOTION refers to the strategy of promoting cached objects only at eviction time. It aims to retain popular objects with minimal effort. An example of LAZY PROMOTION is adding reinsertion to FIFO. In contrast, FIFO has no promotion, and LRU performs eager promotion – moving objects to the head of the queue on every cache hit. LAZY PROMOTION can improve (1) throughput due to less computation and (2) efficiency due to more information about an object at eviction.

QUICK DEMOTION removes most objects quickly after they are inserted. Many previous works have discussed this idea in the context of evicting pages from a scan [38, 172, 227, 263, 291, 309]. Recent work also shows that not only storage workloads but web cache workloads also benefit from QUICK DEMOTION [375] because object popularity follows a power-law distribution, and many objects are unpopular.

To the best of our knowledge, our proposed cache eviction algorithm, which we call SIEVE, is the simplest one that effectively achieves both LAZY PROMOTION and QUICK DEMOTION.

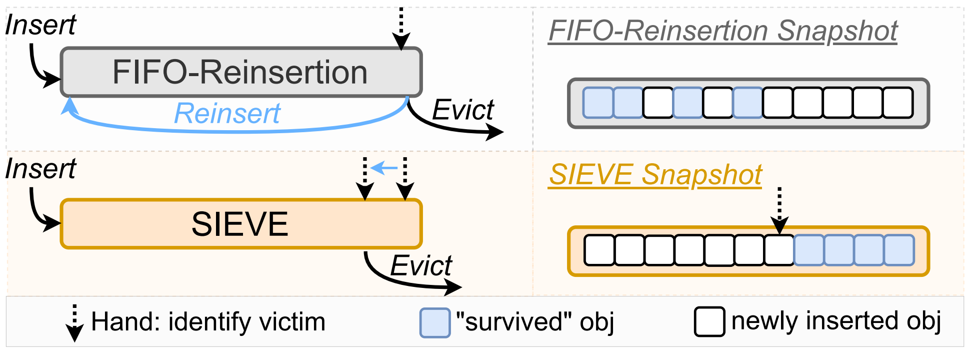


Figure 9.1: An illustration of Sieve. Note that FIFO-Reinsertion and CLOCK are different implementations of the same algorithm. We use FIFO-Reinsertion in the illustration but will use CLOCK in the rest of the text because it is more commonly used and is shorter.

9.2 Design and Implementation

9.2.1 Sieve Design

In this section, we introduce Sieve, a cache eviction algorithm that achieves both simplicity and efficiency.

Data structure. Sieve requires only one FIFO queue and one pointer called “hand”. The queue maintains the insertion order between objects. Each object in the queue uses one bit to track the visited/non-visited status. The hand points to the next eviction candidate in the cache and moves from the tail to the head. Note that, unlike existing algorithms, e.g., LRU, FIFO, and CLOCK, in which the eviction candidate is always the tail object, the eviction candidate in Sieve is an object somewhere in the queue.

Sieve operations. A cache hit in Sieve changes the visited bit of the accessed object to 1. For a popular object whose visited bit is already 1, Sieve does not need to perform any operation. During a cache miss, Sieve examines the object pointed by the hand. If it has been visited, the visited bit is reset, and the hand moves to the next position (the retained object stays in the original position of the queue). It continues this process until it encounters an object with the visited bit being 0, and it evicts the object. After the eviction, the hand points to the next position (the previous object in the queue). While an evicted object is in the middle of the queue most of the time, a new object is always inserted into the head of the queue. In other words, the new objects and the retained objects are not mixed together.

At first glance, Sieve is similar to CLOCK/Second Chance/FIFO-Reinsertion [1]. Each algorithm maintains a single queue in which each object is associated with a visited bit to track its access status. Visited objects are retained (also called "survived") during an eviction. Notably, new objects are inserted at the head of the queue in both Sieve and FIFO-Reinsertion. However, the hand in Sieve moves from the tail to the head over time, whereas the hand in FIFO-Reinsertion stays at the tail. The key difference is where a retained object is kept. Sieve keeps it in the old position, while FIFO-Reinsertion inserts it at the head, together with newly inserted objects, as depicted in Figure 9.1.

We detail the algorithm in Alg. 6. Line 1 checks whether there is a hit, and if so, then line 2 sets the visited bit to one. In the case of a cache miss (Line 3), Lines 5-12 identify the object to be evicted.

lazy promotion and quick demotion. Despite a simple design, Sieve effectively incorporates both lazy promotion and quick demotion. As described in section 9.1, an object is only promoted at the eviction time in lazy promotion. Sieve operates in a similar manner. However, rather than promoting the object to the head of the queue, Sieve keeps the object at its original location. The "survived" objects are generally more popular than the evicted ones, thus, they are likely to be accessed again in the future. By gathering the "survived" objects, the hand in Sieve can quickly move from the tail to the area near the head, where most objects are newly inserted. These newly inserted objects are quickly examined by the hand of Sieve after they are admitted into the cache, thus achieving quick demotion. This eviction mechanism makes Sieve achieve both lazy promotion and quick demotion without adding too much overhead.

The key ingredient of Sieve is the moving hand, which functions like an adaptive filter that removes unpopular objects from the cache. This mechanism enables Sieve to strike a balance between finding new popular objects and keeping old popular objects. We discuss more in section 9.4.

[1]Note that Second Chance, CLOCK, and FIFO-Reinsertion are different implementations of the same eviction algorithm.

Algorithm 6 SIEVE

```
Require: The request $x$, doubly-linked queue $T$, cache size $C$, hand $p$
 1: if $x$ is in $T$ then                                    ▷ Cache Hit
 2:     $x$.visited $\leftarrow$ 1
 3: else                                                     ▷ Cache Miss
 4:     if $|T| = C$ then                                    ▷ Cache Full
 5:         $o \leftarrow$ p
 6:         if $o$ is NULL then
 7:             $o \leftarrow$ tail of $T$
 8:         end if
 9:         while $o$.visited = 1 do
10:             $o$.visited $\leftarrow$ 0
11:             $o \leftarrow o$.prev
12:             if $o$ is NULL then
13:                 $o \leftarrow$ tail of $T$
14:             end if
15:         end while
16:         $p \leftarrow o$.prev
17:         Discard $o$ in $T$                               ▷ Eviction
18:     end if
19:     Insert $x$ in the head of $T$.
20:     $x$.visited $\leftarrow$ 0                           ▷ Insertion
21: end if
```

9.2.2 Implementation

Simulation. We implemented SIEVE in libCacheSim [368]. LibCacheSim is a high-performance cache simulator designed to run cache simulations and analyze cache traces. It supports many state-of-the-art eviction algorithms, including ARC [227], LIRS [165], CACHEUS [291], LeCaR [329], TwoQ [172], LHD [38], Hyperbolic [48], FIFO-Reinsertion/CLOCK [90], B-LRU (Bloom Filter LRU), LRU, LFU, and FIFO. For all state-of-the-art algorithms, we used the configurations from the original papers.

Prototype. Because of SIEVE's simplicity, it can be implemented by changing a few lines from a FIFO, LRU, or CLOCK cache to add, initialize, and track the "hand" pointer. The object pointed to by the hand is either evicted or retained, depending on whether it has been accessed.

We implemented SIEVE caching in five different open-source cache libraries: Cachelib [41], groupcache [50], mnemonist [247], lru-dict [213], and lru-rs [214]. These represent the most popular cache libraries of five different programming languages: C++, Golang, JavaScript, Python, and Rust. All five of these production cache libraries implement LRU as the eviction algorithm of choice. Aside from mnemonist, which uses arrays, they all use doubly-linked-list-based implementations of LRU. Adapting these LRU implementations to use SIEVE was a low effort, as mentioned earlier.

9.3 Evaluation

In this section, we evaluate SIEVE to answer the following questions.

- Does SIEVE have higher efficiency than state-of-the-art cache eviction algorithms?
- Can SIEVE improve a cache's throughput and scalability?
- Is SIEVE simpler than other algorithms?

9.3.1 Experimental setup

Workloads. Our experiments use open-source traces from Twitter [370], Meta [3], Wikimedia [349], TencentPhoto [394, 395], and two proprietary CDN datasets. We list the dataset information in Table 9.1. It consists of 1559 traces that together contain 247,017 million requests to 14,852 million objects. Notably, our research is centered around web traces. We replayed the traces in the simulator and the prototypes as a closed system with instant on-demand fill.

Metrics. Miss ratio serves as a key performance indicator when evaluating the efficiency of a cache system. However, when analyzing different traces (even within the same dataset), the miss ratios can vary significantly, making direct comparisons and visualizations infeasible, as shown in Figure 9.2. Therefore, we calculate the miss ratio reduction relative to a baseline method (FIFO in this work): $\frac{mr_{FIFO} - mr_{algo}}{mr_{FIFO}}$ where mr stands for miss ratio. If an algorithm's miss ratio is higher than FIFO, we use $\frac{mr_{FIFO} - mr_{algo}}{mr_{algo}}$. This metric has a range between -1 and 1.

We measure throughput in millions of operations per second (Mops) to quantify a

Table 9.1: Datasets used in this work. CDN 1 and 2 are proprietary, and all others are publicly available.

trace collections	approx time	# traces	cache type	# request (million)	# object (million)
CDN 1	2021	1273	object	37,460	2,652
CDN 2	2018	219	object	3,728	298
Tencent Photo [394]	2018	2	object	5,650	1,038
Wiki CDN [349]	2019	3	object	2,863	56
Twitter KV [370]	2020	54	KV	195,441	10,650
Meta KV [3]	2022	5	KV	1,644	82
Meta CDN [3]	2023	3	object	231	76

cache's performance. To evaluate scalability, we vary the number of trace replay threads from 1 to 16 and measure the throughput.

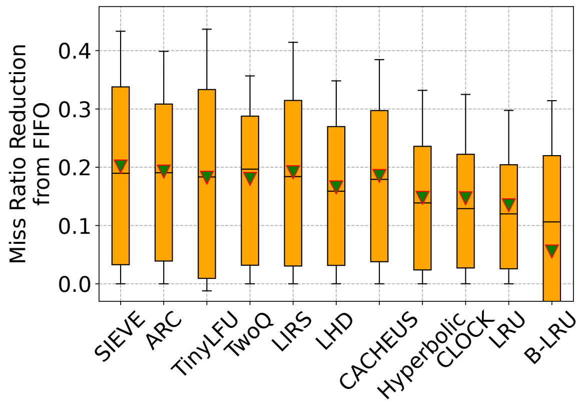
Testbed. Our evaluations were conducted on Cloudlab [107] and focused on two key aspects: simulation-based efficiency and prototype-based throughput and simplicity.

We used libCacheSim [368], a high-performance cache simulator, to evaluate the efficiency of different cache algorithms. These simulations ran on various node types at either the Clemson or Utah sites, subject to availability.

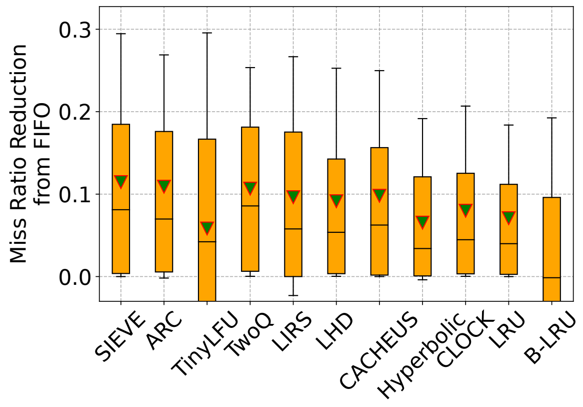
We evaluate the throughput and simplicity using prototypes, as described in subsection 9.2.2. The prototype evaluations were conducted on the c6420 node from the Clemson site. This node type has a dual-socket Intel Gold 6142 running at 2.6 GHz and is equipped with 384 GB DDR4 DRAM. We turned off turbo boost and pinned threads to CPU cores in one NUMA node in our evaluations. We validated the efficiency results from the simulator and prototype using 60 randomly selected traces and found the same conclusion.

9.3.2 Efficiency results

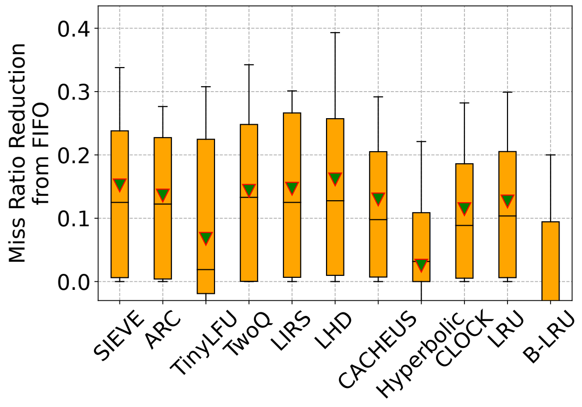
In this section, we compare the efficiency of different eviction algorithms. Because many caches today use slab-based space management, in which evictions happen on objects of similar sizes, we do not consider object size in this section. The cache sizes are determined



(a) CDN1 workloads, large cache, 1273 traces



(b) CDN2 workloads, large cache, 219 traces

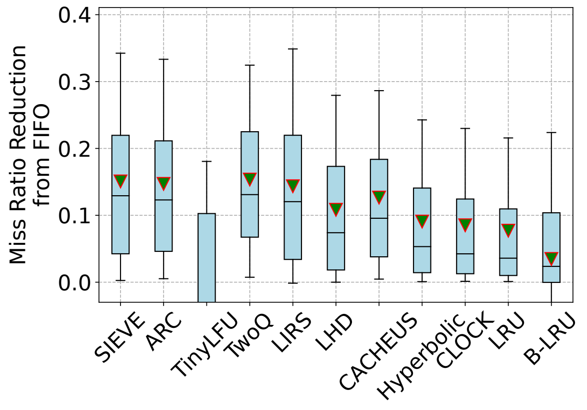


(c) Twitter workloads, large cache, 54 traces

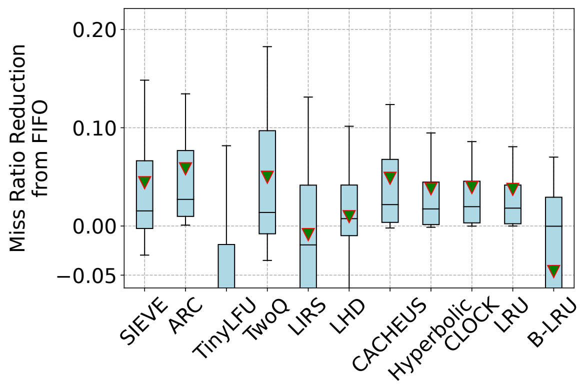
Figure 9.2: The miss ratio reduction from FIFO over all traces in the dataset. This shows the large cache using 10% of the trace footprint.

as a percentage of the number of objects in a trace. We evaluate eight cache sizes using 1559 traces from the 7 datasets and present two representative cache sizes at 0.1% and 10% of the trace footprint (the number of unique objects in the trace).

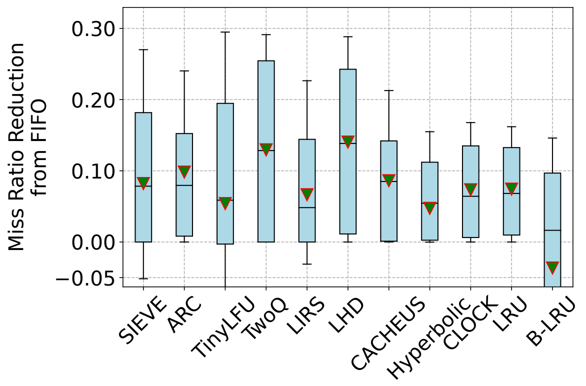
Three large datasets CDN1, CDN2 and Twitter. Figure 9.2 and Figure 9.3 show the miss ratio reduction (from FIFO) of different algorithms across traces. The whiskers on the boxplots are defined using p10 and p90, allowing us to disregard extreme data and concentrate on the typical cases. At the large cache size, Sieve demonstrates the most significant reductions across nearly all percentiles. For example, Sieve reduces FIFO's miss ratio by more than 42% on 10% of the traces (top whisker) with a mean of 21% on the CDN1 dataset using the large cache size (Figure 9.2a). As a comparison, all



(**a**) CDN1 workloads, small cache, 1273 traces



(**b**) CDN2 workloads, small cache, 219 traces

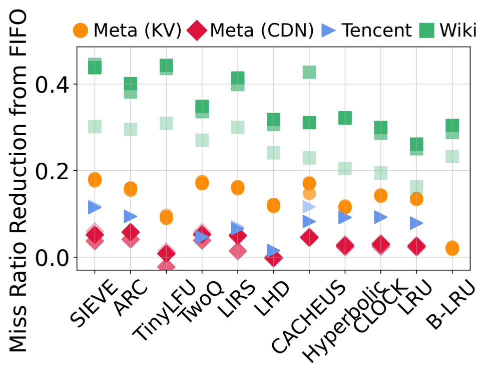


(**c**) Twitter workloads, small cache, 54 traces

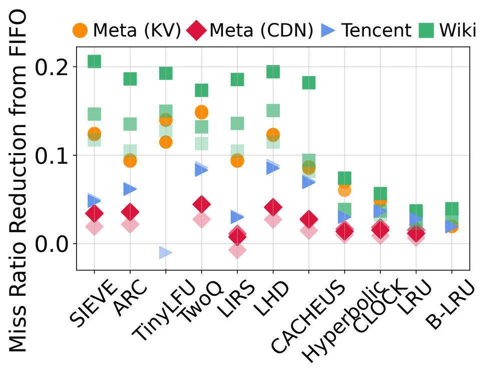
Figure 9.3: The miss ratio reduction from FIFO over all traces in the dataset. This shows the small cache using 0.1% of the trace footprint.

other algorithms have smaller reductions on this dataset. For example, CLOCK/FIFO-Reinsertion, which is conceptually similar to Sieve, can only reduce FIFO's miss ratio by 15% on average. Compared to advanced algorithms, e.g., ARC, Sieve reduces ARC miss ratio by up to 63.2% with a mean of 1.5%. We remark that a 1.5% mean miss ratio reduction on the huge number of traces is significant. For example, ARC only reduces LRU's miss ratio by 6.3% on average (not shown). A similar observation can be made on the CDN2 dataset. Although LHD is the best algorithm on the Twitter dataset, Sieve scores second and outperforms most other state-of-the-art algorithms.

When the cache is very small, TwoQ and LHD sometimes outperform Sieve. This is



(**a**) Large cache



(**b**) Small cache

Figure 9.4: Miss ratio reduction on Meta (KV + CDN), Wiki CDN, and Tencent Photo CDN datasets. The different opacity of the same color indicates multiple traces from the dataset. Some negative results are not shown.

because TwoQ and LHD can quickly remove newly-inserted low-value objects similar to SIEVE. The primary reason for SIEVE's relatively poor performance is that new objects cannot demonstrate their popularity before being evicted when the cache size is very small. A similar problem also happens with ARC and LIRS. ARC's adaptive algorithm sometimes shrinks the recency queue to very small and yields a high miss ratio. LIRS, which uses a 1% queue for new objects, suffers the most when the cache size is small, as we see its miss ratio on some traces higher than FIFO. In contrast, TwoQ does not suffer from the small cache sizes because it reserves a fixed 25% of the cache space for new objects, preventing overly aggressive demotion. However, we remark that the production miss ratios reported in previous works [25, 154, 370, 372] are close to the miss ratios we observe at the large cache size.

The secret behind SIEVE's efficiency is the ability to quickly remove newly-inserted unpopular objects (QUICK DEMOTION), the ability to sift out old unpopular objects, and the balance between new and old objects. We discuss more in section 9.4.

Four small datasets: Meta KV, Meta CDN, Wiki, and TencentPhoto. Because each dataset contains fewer than ten traces, we use scatter plots to compare the algorithms. Figure 9.4 demonstrates that SIEVE outperforms all other algorithms on all four datasets at the large cache size. When the cache size is small, the observation is similar to that made in Figure 9.2. SIEVE is the best algorithm on the Wiki dataset. TwoQ and LHD are the best on Meta and TencentPhoto datasets. Although not the best, SIEVE remains highly

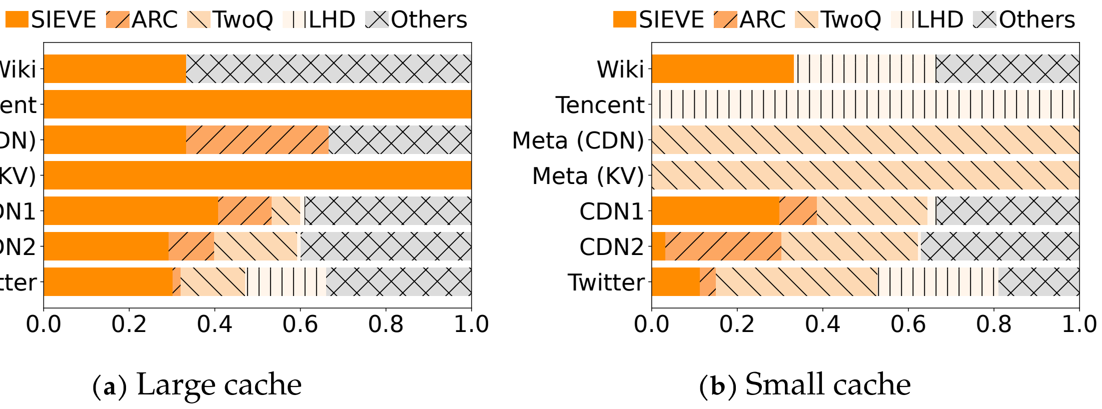


Figure 9.5: Best-performing algorithms on each dataset. Table 9.1 shows the number of traces per dataset.

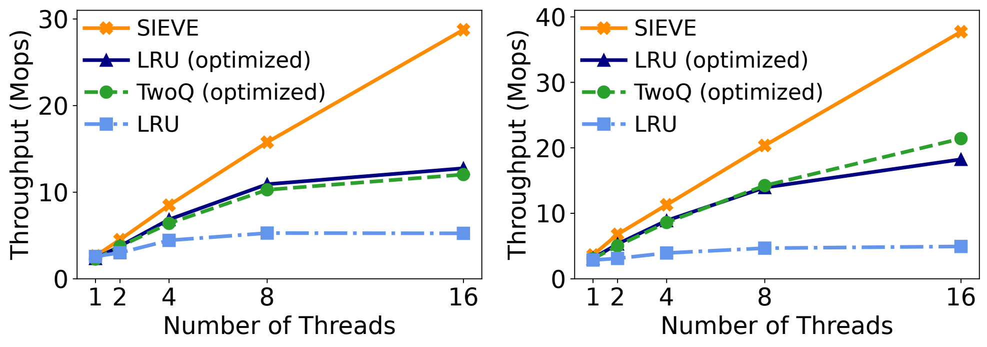
competitive.

Best-performing algorithm per dataset. We have demonstrated that Sieve provides larger miss ratio reductions across traces than state-of-the-art algorithms. For a more quantitative comparison, Figure 9.5 shows the fraction of traces each algorithm performs the best.

With a large cache size, Sieve outperforms all other algorithms on the Tencent Photo, Wiki, and Meta KV datasets. On the CDN1 and CDN2 datasets, Sieve is the best algorithm on 48% and 38% of the 1273 and 219 traces. On the Twitter dataset, although Sieve is the best on only 30% of the traces, it is important to note that no other algorithms are the best on more than 18% of the traces. When using the small cache size, Sieve, TwoQ is the best algorithm winning on the two Meta datasets. On the other datasets, Sieve and LHD have similar shares being the best-performing algorithms. The reason for the observation is similar to that previously explained.

9.3.3 Throughput performance

Besides efficiency, throughput is the other important metric for caching systems. Although we have implemented Sieve in five different libraries, we focus on Cachelib's results. Because all other libraries implement strict LRU and do not consider object sizes, evaluations yield the same miss ratio as our simulation. Moreover, strict LRU is not scalable, as we show next.



(**a**) Meta KV trace (**b**) Twitter trace

Figure 9.6: Throughput scaling with CPU cores on two KV-cache workloads.

Figure 9.6 shows how throughput grows with the number of trace replay threads using two production traces from Meta and Twitter. To better emulate real-world deployments in which the working set size (dataset size) grows with the hardware specs (#cores and DRAM sizes), we scale the cache size and working set size together with the number of threads. To scale the working set size, each thread plays the same trace with the object id transformed into a new space. For example, the benchmark sends $4\times$ more requests to $4\times$ larger cache size at 4 threads compared to the single-thread experiment. We set the cache size to be $4 \times n_{thread}$ GB for both traces, which gives miss ratios of 7% (Meta) and 2% (Twitter). We remark that the miss ratio is close to previous reports [25, 372].

The LRU and TwoQ in Cachelib use extensive optimizations to improve the scalability. For example, objects that were promoted to the head of the queue in the last 60 seconds are not promoted again, which reduces lock contention without compromising the miss ratio. Cachelib further adds a lock combining technique to elide expensive coherence and synchronization operations to boost throughput [96]. As a result of the optimizations, both LRU and TwoQ show impressive scalability results compared to the unoptimized LRU: the throughput is $6\times$ higher at 16 threads than using a single thread on the Twitter trace. As a comparison, unoptimized LRU's throughput plateaus at 4 threads.

Compared to these LRU-based algorithms, SIEVE does not require "promotion" at each cache hit. Therefore, it is faster and more scalable. At a single thread, SIEVE is 16% (17%) faster than the optimized LRU (TwoQ) and on both traces. At 16 threads, SIEVE shows more than $2\times$ higher throughput than the optimized LRU and TwoQ on the Meta trace.

Table 9.2: Lines of code modification required to add SIEVE to a production cache library.

Cache library	Language	Lines
groupcache [50]	Golang	21
mnemonist [247]	Javascript	12
lru-rs [214]	Rust	16
lru-dict [213]	Python + C	21

Table 9.3: Lines of code (excluding comments and empty lines) and per-object metadata size required to implement each algorithm in our simulator. We assume that frequency counter and timestamps use 4 bytes and pointers use 8 bytes.

Algorithm	cache hit	eviction	insertion	metadata size
FIFO	1	4	3	16B
LRU	5	4	3	16B
ARC	64	108	20	17B
LIRS	96	120	64	17B
LHD	192	81	64	13B
LeCaR	72	76	20	40B
CACHEUS	168	140	150	54B
TwoQ	28	16	8	17B
Hyberbolic	4	20	4	16B
CLOCK	4	9	3	17B
SIEVE	4	9	3	17B

9.3.4 Simplicity

Prototype implementations. SIEVE not only achieves better efficiency, higher throughput, and better scalability, but it is also very simple. We chose the most popular cache libraries/systems from five different languages: C++, Go, JavaScript, Python, and Rust, and replaced the LRU with SIEVE.

Although different libraries/systems have different implementations of LRU, e.g., most use doubly-linked-list, and some use arrays, we find that implementing SIEVE is very easy. Table 9.2 shows the number of lines (not including the tests) needed to replace LRU — all implementations require no more than 21 lines of code changes [2].

[2]While most LRU implementations are straightforward to adapt for SIEVE, CacheLib is an exception.

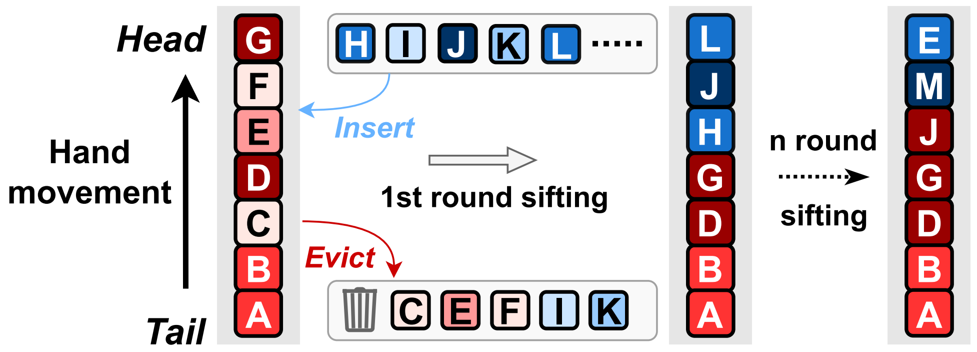
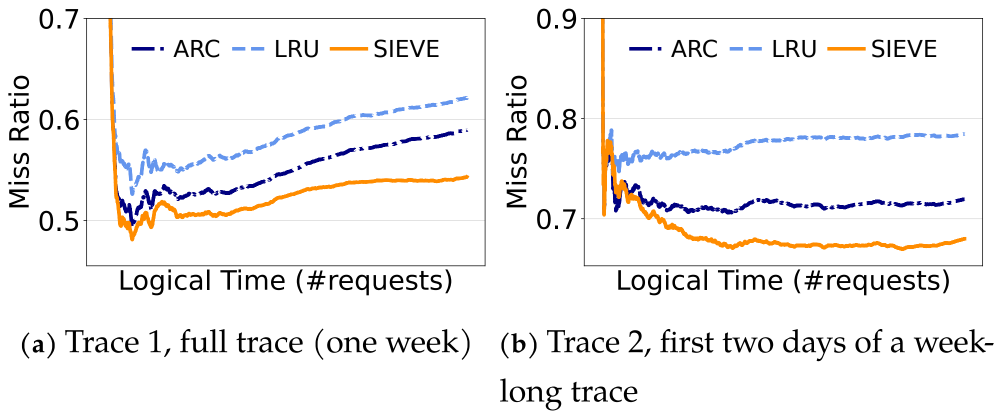


Figure 9.7: Density of colors indicates inherent object popularity (blue: newly inserted objects; red: old objects in each round), and the letters represent object IDs. The first queue captures the state at the start of the first round, and the second queue captures the state at the end of the first round.



(a) Trace 1, full trace (one week) (b) Trace 2, first two days of a week-long trace

Figure 9.8: Left: illustration of the sifting process. Right: Miss ratio over time for two traces. The gaps between SIEVE's miss ratio and others enlarge over time.

Advanced algorithms in simulator. Most of the complex algorithms we evaluated in subsection 9.3.2 are not implemented in production systems. Therefore, we compare the lines of code needed to implement cache hit, insert, and evict in our simulator. Although we implemented our own linked list and hash table data structures in C for our simulator, we do not include the code lines related to list and hash table operations, i.e., appending to the list head or inserting to the hash table requires one line.

Cachelib is highly optimized for LRU-based algorithms. Many optimizations are not needed for SIEVE, making it impractical to quantify code modifications for integration with SIEVE. Therefore, it is not included in Table 9.2.

Table 9.3 shows that FIFO requires the fewest number of lines to implement. On top of FIFO, implementing LRU adds a few lines to promote an object upon cache hits. CLOCK and Sieve require close to 10 lines to implement the eviction function because both need to find the first object that has not been visited. However, we remark that Sieve is simpler than LRU and CLOCK because Sieve does not require moving objects to the head of the queue in either hit or miss (evict). Besides these, all other algorithms require one to two orders more lines of code to implement the three functions.

Per-object metadata. In addition to the implementation complexity, we also quantified the per-object metadata needed to implement each algorithm. FIFO does not require any metadata when implemented using a ring buffer. However, such an implementation does not support overwrite or delete. So common FIFO implementation also uses a doubly-linked list with 16 bytes of per-object metadata similar to LRU. CLOCK and Sieve are similar, both requiring 1-bit to track object access status. When implemented using a doubly linked list, they use 17 bytes per-object metadata. Compared to Sieve, advanced algorithms often require more per-object metadata. Many key-value cache workloads have objects as small as 10s of bytes [226, 370], and large metadata wastes the precious cache space.

ZERO parameter. *Besides being easy to implement and having less metadata, Sieve also has no parameters*. Except for FIFO, LRU, CLOCK, and Hyperbolic, all other algorithms have explicit or implicit parameters, e.g., the sizes of queues in LIRS, the learning rate in LeCaR and CACHEUS, and the decay rate and age granularity in LHD. Note that although ARC has no explicit parameters, its adaptive algorithm uses implicit parameters in deciding when and how much space to move between the queues. As a comparison, Sieve has no parameter and requires no tuning.

9.4 Distilling Sieve's Effectiveness

Our empirical evaluation shows that Sieve is simultaneously simple, fast, scalable, and efficient. In a well-trodden field like cache eviction, Sieve's competitive performance was a genuine surprise to us as well. We next report our analysis that seeks to understand the secrets behind its efficiency.

9.4.1 Visualizing the sifting process

The workhorse of SIEVE is the "hand" that functions as a sieve: it sifts through the cache to filter out unpopular objects and retain the popular ones. We illustrate this process in Figure 9.7, where each column (queue) represents a snapshot of the cached objects over time from left to right. As the hand moves from the tail (the oldest object) to the head (the newest object), objects that have not been visited are evicted – the same sweeping mechanism that underlies CLOCK [67, 90]. For example, after the first round of sifting, objects at least as popular as A remain in the cache while others are evicted. The newly admitted objects are placed at the head of the queue — much like the CLOCK policy, but a departure from CLOCK, which does in-place replacements to emulate LRU. During the subsequent rounds of sifting, if objects that survived previous rounds remain popular, they will stay in the cache. In such a case, since most old objects are not evicted, the eviction hand quickly moves past the old popular objects to the queue positions close to the head. This allows newly inserted objects to be quickly assessed and evicted, putting greater eviction pressure on unpopular items (such as "one-hit wonders") than LRU and its variations [227]. As previous work has shown [38, 154, 375], QUICK DEMOTION is crucial for achieving high cache efficiency.

Figure 9.8a and Figure 9.8b show the cumulative miss ratio over time of different algorithms on two representative production traces. After the cache is warmed up, the miss ratio gaps between SIEVE and other algorithms widen over time, supporting the interpretation that SIEVE indeed sifts out unpopular objects and retains popular ones. A similar observation can be seen in Figure 9.11a.

9.4.2 Analyzing the sifting process

We now analyze the popularity retention mechanism in SIEVE. To clarify the exposition, suppose the SIEVE cache can fit C equally sized objects. Since SIEVE always inserts new objects at the head, and objects that are retained remain in their original positions within the queue, the algorithm implicitly partitions the cache between new and old objects. This partition is dynamic, allowing SIEVE to strike a balance between exploration (finding new popular objects) and exploitation (enjoying hits on old popular objects).

SIEVE performs sifting by moving the hand from the tail to the head, evicting unpopular objects along the way, which we call one round of sifting. We use r to denote the number of rounds. We first enumerate the queue positions p from the tail ($p = 0$) to the head ($p = C - 1$). We then further denote that an object at position p in round r is *examined* (during eviction) or *inserted* at time T_p^r. Note that T effectively defines a logical timer for the examined objects: whenever an object is examined, T increases by 1, regardless of whether the examined object is evicted or retained. In addition, T changes *once* each round for an old object (retained from previous rounds).

For an old object x at position p, we define the "inter-examination time" $I_e(p^r) = T_p^r - T_{p'}^{r-1}$ where p' was the position of x in round $r-1$. Clearly, $p' \geq p$. For a new object inserted in the current round, the inter-examination time is defined as the time between its examination and insertion. We further define an old object x's "inter-arrival time" $I_a(x^r)$ as the time, measured again in the number of objects examined, between the first request to the x in round r and the last request to x in round $r-1$. For a new object, the inter-arrival time is the time between its insertion and the second request. If an old object is not requested in the last round or a new object does not have a second request, its inter-arrival time is infinite.

In round r, consider two consecutive retained objects x_1 and x_2 at position p_1 and $p_2 = p_1 + 1$. The inter-examination times are $I_e(p_1^r) = T_{p_1}^r - T_{p_1'}^{r-1}$ and $I_e(p_2^r) = T_{p_2}^r - T_{p_2'}^{r-1}$, respectively. The transition yields two invariants:

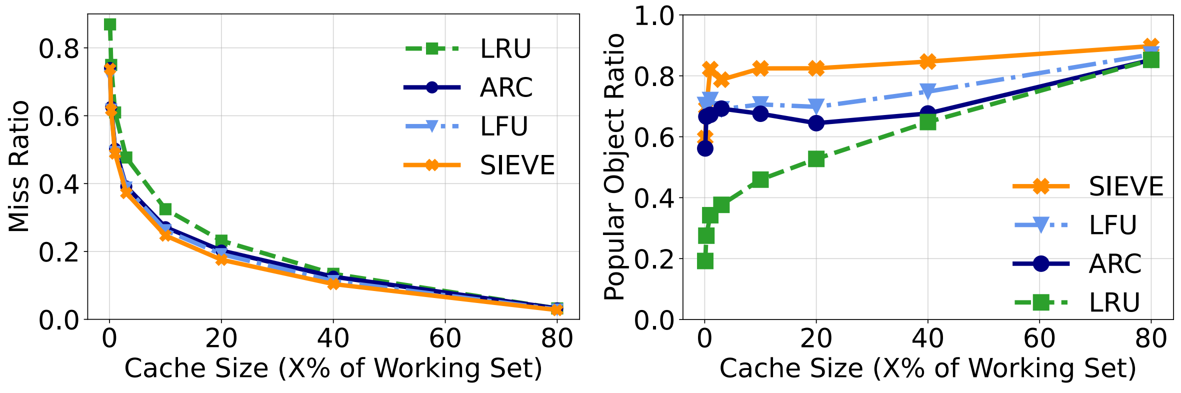
$$\begin{aligned} T_{p_2}^r - T_{p_1}^r &= 1 \\ T_{p_2'}^{r-1} - T_{p_1'}^{r-1} &\geq 1 \end{aligned}$$

The first equation follows from x_1 and x_2 being consecutively retained objects; the second inequality expresses that other evictions may have taken place between x_1 and x_2 in the previous round. Together, these imply that $I_e(p_1^r) \geq I_e(p_2^r)$. The result generalizes further: for any two retained old objects in the queue, the object closer to the head has a smaller inter-examination time.

Moreover, if an object is retained, its inter-arrival time must be no greater than its inter-examination time. Therefore, for any retained object x at position p_x, its inter-arrival time $I_a(x^r)$ must be smaller than the tail object's inter-examination time:

$$I_a(x^r) \leq I_e(p_x^r) \leq I_e(p_0^r) \tag{9.1}$$

Using the commonly assumed independent reference model [74, 122, 157, 158] with a Poisson arrival, we can expect any retained object to be more popular than some dynamic



(**a**) Miss ratio over size (**b**) Popular object ratio over size

Figure 9.9: Miss ratio and popular object ratio on a Zipfian dataset $(\alpha = 1.0)$.

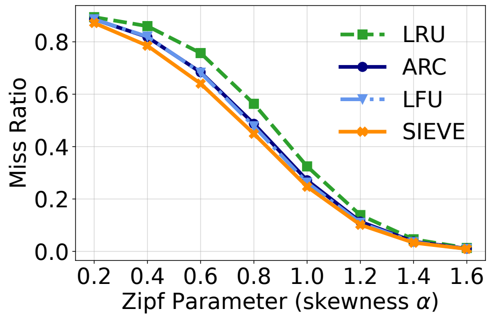
threshold set by the tail object's inter-examination time $I_e(p_0^r)$. Since evicting an object keeps the hand pointer at its original position (relative to the tail), the more objects are evicted during a round, the longer the inter-examination time. As a result, SIEVE effectively adapts the popularity threshold so that more objects are retained in the next round.

Following our sifting process metaphor, the mesh size in SIEVE is determined by the tail object's inter-examination time $I_e(p_0^r)$, which is dynamically adjusted based on object popularity change. If too few objects are retained in one round (mesh size too small), then we will have an increased tail inter-examination time $I_e(p_0^r)$ (a larger mesh size) in the next round.

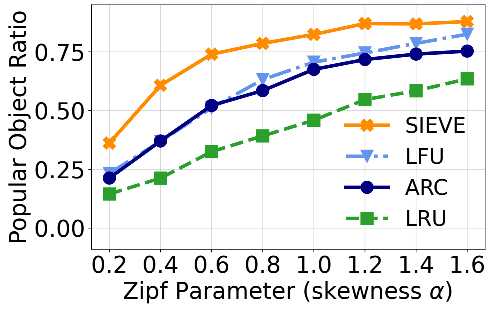
9.4.3 Deeper study with synthetic workloads

Production trace workloads are often too complex and dynamic to analyze. One consistent finding from past workload characterization work, however, is that object popularity in web cache workloads invariably follows a heavy-tailed power-law (generalized Zipfian) distribution [52, 370]. Therefore, we opted for synthetic power-law workloads for our study. It allows us to easily modify workload features to better understand their impact on performance. Using these synthetic workloads, we further scrutinize SIEVE's effectiveness.

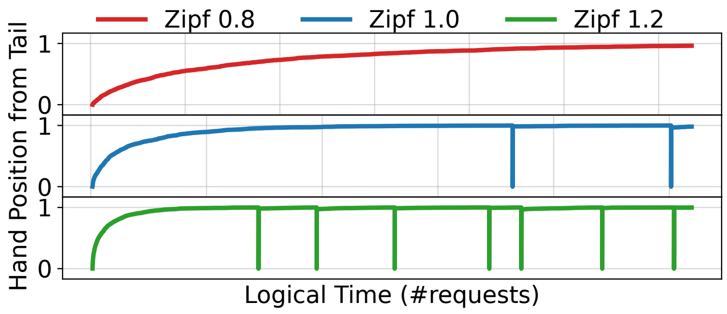
Miss ratio over size. Figure 9.9a displays the miss ratio of LRU, LFU, ARC, and SIEVE at



(**a**) Miss ratio



(**b**) Popular object ratio



(**c**) Hand movement in SIEVE

Figure 9.10: Left two: miss ratio and popular object ratio on Zipfian workloads with different α. Right: hand position in the cache over time in Zipfian workloads.

different cache sizes. Notably, LFU, ARC, and SIEVE all exhibit lower miss ratios than LRU, demonstrating their efficiency. Despite being considered optimal for synthetic power-law workloads, LFU performs similarly to ARC and is visibly worse than SIEVE. This is because objects with medium popularity, such as objects with ranks around the cache size C, are only requested once before their eviction. LFU cannot distinguish the true popularity of these objects and misses out on an opportunity for better performance. As a comparison, both ARC and SIEVE can quickly remove new and potentially unpopular objects, which allows cached objects to enjoy more time in the cache to demonstrate their popularity. Between the two algorithms, SIEVE further extends the tenure of these objects in the cache because when the hand sweeps through the newly inserted objects, the objects closer to the head must have strictly shorter inter-arrival times (expected to be more popular) to survive.

Popular object ratio over size. To capture how different algorithms manage popular objects, we define a metric called "popular object ratio". Under the assumption of a static and known popularity distribution, the optimal caching policy retains the most popular content within the cache at all times. Given a cache size C and a workload following a power-law distribution, the popular objects are the C most frequent objects in the workload, denoted by H. The popular ratio of objects in the cache at time t is calculated by $I_t = \frac{|H \cap A_t|}{C}$ where A_t denotes the cache contents at time t.

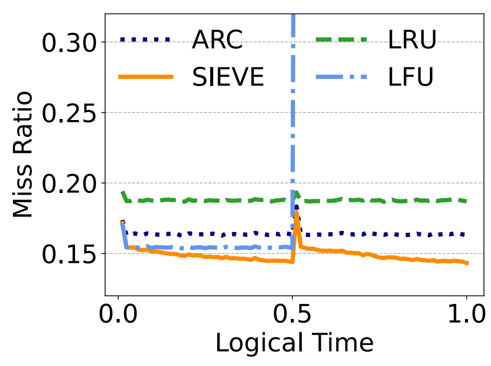
Figure 9.9b shows the popular object ratio at different cache sizes. LRU evicts objects based on recency, which only weakly correlates with popularity. In this scenario, LRU stores the least number of popular objects. LFU stores slightly more "popular objects" than ARC. SIEVE, however, successfully filters out unpopular objects from the cache.

Varying the popularity skew. Figure 9.9 shows a distribution with Zipfian skewness $\alpha = 1$. We further studied how different concentration of popularity affects SIEVE's effectiveness. Due to space restrictions, we focus on results with large cache sizes for the remainder of this subsection. Results using the small cache size are either similar or do not reveal interesting patterns.

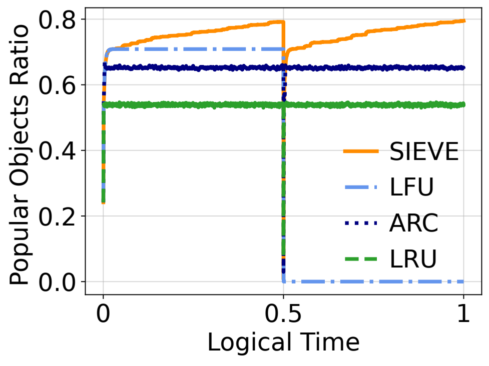
Figure 9.10a and Figure 9.10b demonstrate the impact of varying skew on miss and popular object ratios. As skew increases, making popular objects more prominent, it becomes easier to identify and cache the popular objects, increasing the popular object ratio and reducing the miss ratio for all tested algorithms. Among ARC, LFU, and SIEVE, we observe that SIEVE always shows a higher popular ratio with a lower miss ratio across skewness, indicating the efficiency of SIEVE is not limited to very skewed workloads.

Figure 9.10c illustrates the hand position in the SIEVE cache over time, advancing towards the head with each retained object and pausing during evictions. Therefore, the more objects are retained, the faster the movement. We observe that the hand moves more slowly in the first round than in the later rounds because that is when many unpopular objects are evicted. In subsequent rounds, the hand lingers at positions close to the head for most of the time because SIEVE keeps a new object at position p only if it is more popular (shorter inter-arrival time) than the object at position $p-1$. In other words, SIEVE performs QUICK DEMOTION [351].

In more skewed workloads, the hand moves quickly due to early arrival and higher request volumes for popular objects, allowing SIEVE to cache most popular objects by the end of the first round. Consequently, the hand rapidly transitions from tail to head with fewer evictions and spends less time near the head, as new objects are more likely to be retained, hastening its progress. Nevertheless, the time of each round varies depend-



(**a**) Interval miss ratio

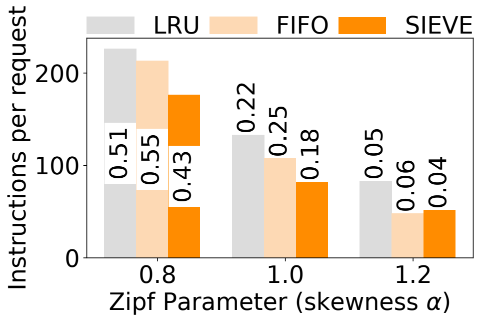


(**b**) Popular object ratio over time

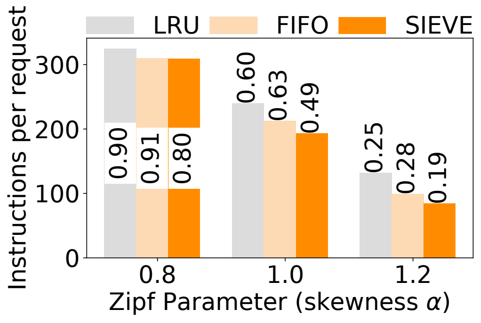
Figure 9.11: Interval miss ratio and popular object ratio over time on a workload constructed by connecting two different Zipfian workloads ($\alpha = 1$).

ing on the frequency of encountering potentially popular objects, highlighting SIEVE's adaptability to workload shifts. When new popular objects appear, the hand accelerates, replacing existing cached objects with the newcomers by giving less time to set their visited bit.

SIEVE is adaptive. To visualize SIEVE's adaptivity via the sifting process, we created a new workload by joining two Zipfian ($\alpha = 1.0$) workloads that request different populations of objects. Figure 9.11 shows the interval miss ratio (per 100,000 requests) over time on this conjoined workload. The changeover happens at the 50% midway time mark. We observe that the interval miss ratio of LFU skyrockets to nearly 100% (beyond figure bounds) since new objects cannot replace the old objects. Relative to LRU and ARC, SIEVE's miss ratio spike is larger because it takes time for the hand to move back to the tail before it can evict old objects. However, SIEVE's spike is invisible when the cache size is small (not shown). With respect to the interval miss ratio spike, we observe the popular object ratio of all algorithms (the curves overlap) dropping to 0 when the workload changes at the midway point. Whereas LFU never recovers from the drop, the popular object ratios in all other algorithms quickly recover to large proportions. Finally, the figures corroborate our interpretation of the sifting process: SIEVE's miss ratio drops over time, while the fraction of popular objects increases over time.



(a) Large cache



(b) Small cache

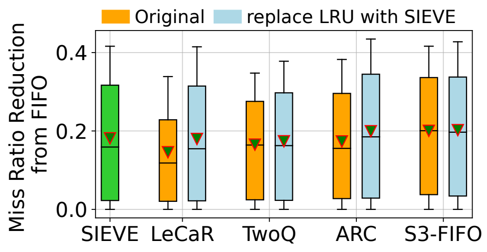
Figure 9.12: Average number of instructions per request when running LRU, FIFO, and SIEVE caches. The top number denotes the miss ratio.

9.5 SIEVE as a Turn-key Cache Primitive

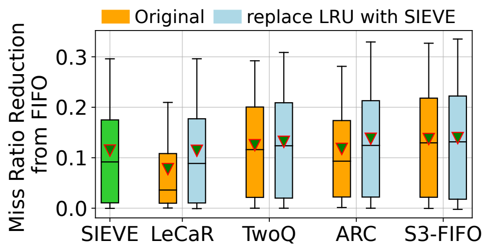
9.5.1 Cache primitives

Beyond being a cache eviction algorithm, SIEVE can serve as a cache primitive for designing more advanced eviction policies. To study the range of such policies, we categorize existing cache eviction algorithm designs into four main approaches. (**1**) We can design simple and easy-to-understand eviction algorithms, such as FIFO queues, LRU queues, LFU queues, and Random eviction. We call these simple algorithms *cache primitives*. SIEVE falls under this category. (**2**) We can improve the cache primitives. For example, FIFO-Reinsertion is designed by adding reinsertion to FIFO; LRU-K [263] is designed by changing the recency metric in LRU. (**3**) We can compose multiple cache primitives with objects moved between them. For example, ARC, SLRU, and MQ use multiple LRU queues. (**4**) We can run multiple cache primitives and craft a decision-maker to select eviction candidates suggested by the primitives. For example, LeCaR [329] uses reinforcement learning to choose between the eviction candidates from LRU and LFU; HALP [313] uses machine learning (MLP) to choose one object from the eight objects at the LRU tail.

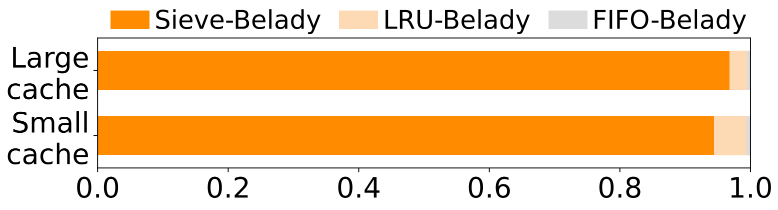
Having an efficient cache primitive not only provides an effective and simple eviction algorithm but also enables other approaches to design more efficient algorithms. The ideal cache primitive is simultaneously (1) simple, (2) efficient, and (3) fast — in terms of high throughput. For example, FIFO and LRU meet these requirements and are



(**a**) Large cache



(**b**) Small cache



(**c**) Best-performing algorithms across traces.

Figure 9.13: Impact of replacing LRU with Sieve in advanced algorithms (**a**,**b**). The potential of FIFO, LRU, and Sieve when endowed with foresight (**c**).

frequently used to construct more advanced algorithms. However, they are less efficient than complex algorithms.

While we have shown that Sieve is simple, efficient, and fast in section 9.3, to further understand Sieve as a cache primitive, we compare the number of instructions needed to run FIFO, LRU, and Sieve caches. We remark that the number of instructions may not necessarily correlate with latency or throughput but rather a rough metric of CPU resource usage. We used `perf stat` to measure the number of instructions for serving power-law workloads (100 million requests, 1 million objects) in our simulator. We then deduct the simulator overhead by measuring a no-op cache, which performs nothing on cache hits and misses.

Figure 9.12 shows that Sieve generally executes fewer instructions per request than FIFO and LRU, a difference accentuated in skewed workloads and larger cache sizes. Compared to LRU, Sieve requires fewer instructions since Sieve needs only to check and possibly update a Boolean field on cache hits, which is much simpler than moving an object to the head of the queue. Besides LRU, Sieve also requires fewer instructions than FIFO because of the difference in miss ratios. Because Sieve has a lower miss ratio than FIFO, fewer objects need to be inserted due to cache misses, leading to fewer instructions per request on average. The only exception is when SIEVE and FIFO have similar miss ratios, in which case, FIFO executes fewer instructions than SIEVE. Overall, SIEVE requires up to 40% and 24% fewer instructions than LRU and FIFO, respectively.

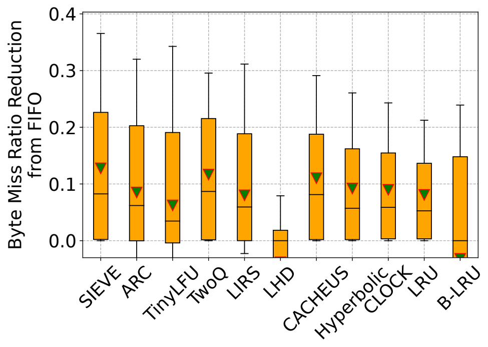
9.5.2 Turn-key cache eviction with SIEVE

As a cache primitive, SIEVE can facilitate the design of more advanced eviction algorithms. To understand the benefits of using a better cache primitive, we replaced the LRU in LeCaR, TwoQ, and ARC with SIEVE. Note that for ARC, we only replace the LRU for frequent objects.

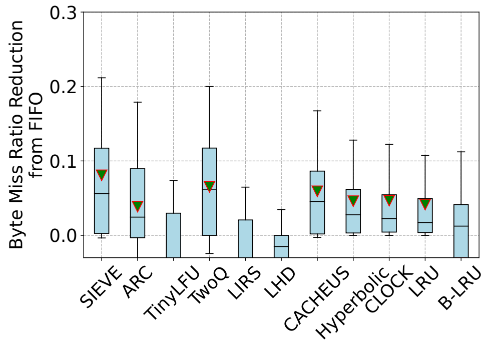
We evaluate these algorithms on all traces and show the miss ratio reduction(from FIFO) in Figure 9.13a and Figure 9.13b. Compared to SIEVE, LeCaR has much lower efficiency; however, when replacing the LRU in LeCaR with SIEVE, it significantly reduces LeCaR's miss ratio by 4.5% on average. TwoQ and ARC achieve efficiency close to SIEVE; however, when replacing the LRU with SIEVE, the efficiency of both algorithms gets boosted. For example, ARC-SIEVE achieves the best efficiency among all compared algorithms at both small and large cache sizes. It reduces ARC's miss ratio by 3.7% on average and up to 62.5% on the large cache size (recall that ARC reduces LRU's miss ratio by 6.3% on average). ARC-SIEVE also reduces SIEVE's miss ratio by an average of 2.4% and up to 40.6%.

To understand the potential in suggesting eviction candidates, we evaluated the efficiency of FIFO, LRU, and SIEVE, granting them access to future request data. Each eviction candidate is either evicted or reinserted, depending on whether the object will be requested soon. We assume that an object will be requested soon if the logical time (number of requests) till the object's next access is no more than $\frac{C}{mr}$, where C is the cache size and mr is the miss ratio. This mimics the case that we have a perfect decision-maker choosing between the eviction candidates suggested by multiple simple eviction algorithms. Figure 9.13c shows that when supplied with this additional information, SIEVE achieves the lowest miss ratio on 97% and 94% of the 1559 traces at the large and small cache size, respectively.

These results highlight the potential of SIEVE as a powerful cache primitive for designing advanced cache eviction algorithms. Leveraging LAZY PROMOTION and QUICK DEMOTION, SIEVE not only performs well on its own but also bolsters the performance of more complex

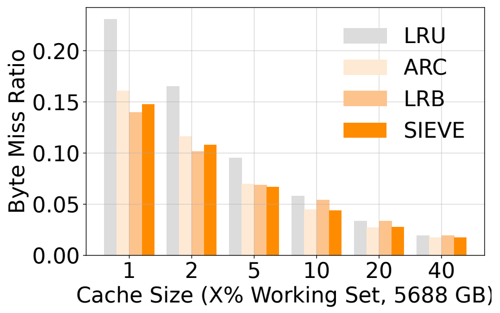


(a) Large cache

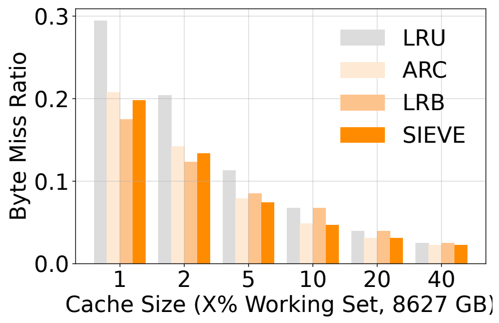


(b) Small cache

Figure 9.14: Byte miss ratio across all CDN traces.



(a) Wiki2018 trace



(b) Wiki2019 trace

Figure 9.15: Byte miss ratios at different cache sizes on two Wiki CDN traces used in LRB evaluation.

algorithms.

9.6 Discussion

9.6.1 Byte miss ratio

To gauge SIEVE's efficiency in reducing network bandwidth usage in CDNs, we analyzed its byte miss ratio by considering object sizes. We chose the cache size at 10% and 0.1% of the trace footprint in bytes. Figure 9.14a and Figure 9.14b show that SIEVE presents larger byte miss ratio reductions at *ALL* percentiles than state-of-the-art algorithms at both cache sizes, showcasing its high efficiency in CDN caches.

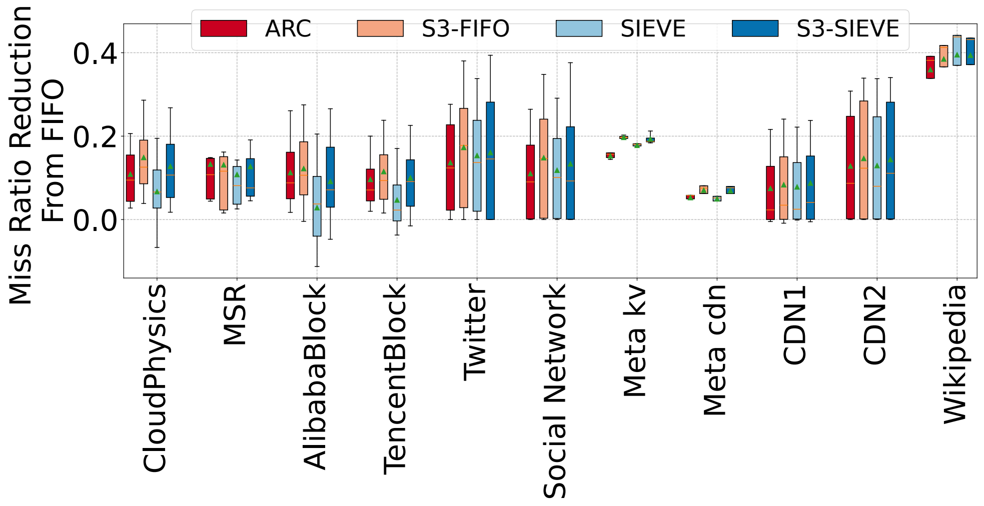
We further compared Sieve with LRB [312], the state-of-the-art machine-learning-based cache eviction algorithm optimized for byte miss ratio. Due to LRB's long run time, we only evaluated LRB on the two open-source Wiki traces provided by the authors. Figure 9.15a and Figure 9.15b show that LRB performs better at small cache sizes (1% and 2%), while Sieve excels at larger cache sizes. We conjecture that at a small cache size, the ideal objects to cache are popular objects with many requests, which LRB can more easily identify because they have more features (most of LRB's features are about the time between accesses to an object). When the cache size is large, most objects in the cache have few requests. Without enough features, a learned model can provide little benefits [375, 376]. In summary, compared to complex machine-learning-based algorithms, Sieve still has competitive efficiency.

9.6.2 Sieve is not scan-resistant

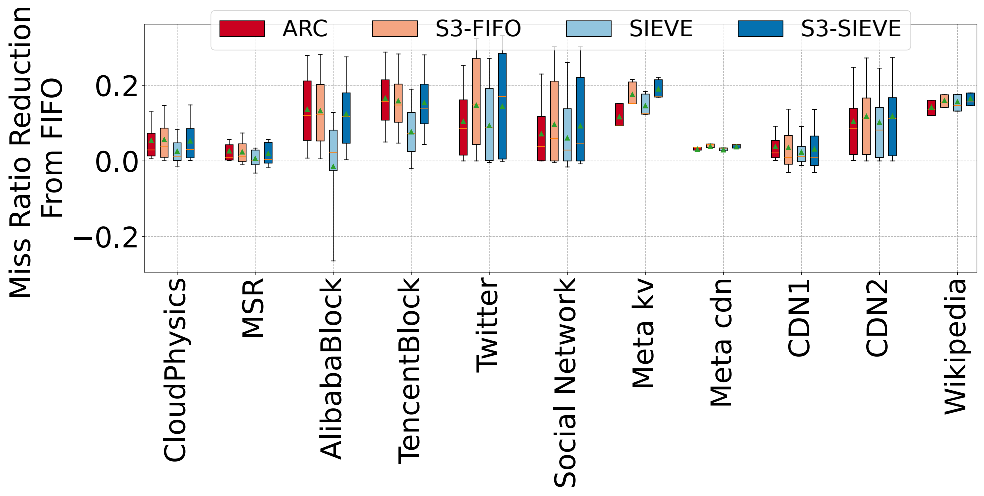
Besides web cache workloads, we evaluated Sieve on some block cache workloads. However, we find that Sieve sometimes shows a miss ratio higher than LRU. The primary reason for this discrepancy is that SIEVE is not scan-resistant. In block cache workloads, which frequently feature scans, popular objects often intermingle with objects from scans. Consequently, both types of objects are rapidly evicted after insertion. Since Sieve does not use a ghost cache — a shadow cache that keeps track of recently evicted items to make smarter future eviction decisions — it cannot recognize the popular objects when they are requested again. This problem is less severe on large caches, but when the cache size is small, we observe that having a ghost is critical to being scan-resistant. We conjecture that not being scan-resistant is probably the reason why Sieve remained undiscovered over the decades of caching research, which has been mostly focused on page and block accesses.

9.6.3 TTL-friendliness

Time-to-live (TTL) is a common feature in web caching [370, 372]. It specifies the duration during which an object can be used. After the TTL has elapsed, the object expires and can no longer be served to the user, even if it may still be cached. Most existing eviction algorithms today do not consider object expiration and require a separate procedure, e.g., scanning the cache, to remove expired objects. Similar to FIFO, Sieve maintains objects in insertion order, which allows objects in TTL-partitioned caches, e.g., Segcache [372], to be sorted by expiration time. This provides a convenient method for discovering and removing expired objects.



(a) Large cache



(b) Small cache

Figure 9.16: Byte miss ratio across all CDN traces.

9.6.4 Comparing S3-FIFO and SIEVE

Both S3-FIFO and SIEVE use simple techniques to achieve LAZY PROMOTION and QUICK DEMOTION. In this section, we compare the efficiency of the ARC, S3-FIFO, SIEVE and S3-SIEVE (using SIEVE as the main cache eviction algorithm in S3-FIFO). Figure 9.16

shows the miss ratio distribution on different datasets.

We observe that SIEVE is visibly worse than both ARC and S3-FIFO on block cache workloads, i.e., CloudPhysics, MSR, AlibabaBlock, and TencentBlock. As explained in Section 9.6.2, SIEVE is not scan-resistant, while block cache workloads often have many scan access patterns, which is why SIEVE shows higher miss ratios on these datasets. When replacing the FIFO-Reinsertion queue in S3-FIFO with SIEVE, the new algorithm S3-SIEVE significantly outperforms SIEVE on these four datasets; however, its miss ratio is still slightly higher than S3-FIFO. This is because the small FIFO queue has filtered out most of the scan requests, which avoids SIEVE's weakness. However, both the small queue and SIEVE perform quick demotion. The combination does not bring extra benefits.

On the web cache workloads, we find that SIEVE achieves similar performance as ARC but still slightly underperforms S3-FIFO, especially at the small cache size. This is because at the small cache size, the QUICK DEMOTION in SIEVE might be too aggressive, and there is no way to control this since SIEVE has no parameter. When equipping S3-FIFO with SIEVE, we find that S3-SIEVE slightly outperforms S3-FIFO on web cache workloads. This could come from the fact that the working set changes quickly in web caches, and the 10% small FIFO queue in S3-FIFO is too large and cannot evict unpopular objects quickly enough.

Overall, SIEVE can achieve state-of-the-art efficiency with a miss ratio close to ARC. However, it is less robust than S3-FIFO and shows a higher miss ratio, especially on block cache workloads. However, the main advantage of SIEVE is the simplicity and scalability that allows it to serve as a primitive.

9.7 Chapter Summary

This chapter describes an easy improvement (**??**) to a decades-old algorithm (FIFO-Reinsertion) that materially improves its efficiency across a wide range of web cache workloads. Instead of moving the to-be-evicted object that has been accessed to the head of the queue, SIEVE keeps it in its original position. It should be noted that both SIEVE and FIFO-Reinsertion insert new objects at the head of the queue. We implemented SIEVE in five production cache libraries, which required fewer than 20 lines of change on average, underscoring the ease of real-world deployment.

Despite a simple design, Sieve can quickly remove unpopular objects from the cache, achieving comparatively high efficiency compared to state-of-the-art algorithms. By experimentally evaluating Sieve on 1559 traces from five public and two proprietary datasets, we show that Sieve achieves similar or higher efficiency than 9 state-of-the-art algorithms across traces. Compared to ARC [227], Sieve reduces miss ratio by up to 63.2% with a mean of 1.5%. As a comparison, ARC reduces LRU's miss ratio by up to 33.7% with a mean of 6.7%. Moreover, compared to the best of all algorithms, Sieve has lower miss ratio on over 45% of the 1559 traces. In comparison, the runner-up algorithm, TwoQ, only outperforms other algorithms on 15% of the traces.

Sieve's design eliminates the need for locking during cache hits, resulting in a boost in multi-threaded throughput. Our prototype implementation in Cachelib [95] demonstrates that Sieve achieves twice the throughput of an optimized LRU implementation when operating with 16 threads.

Through empirical evidence and analysis, we illustrate that Sieve's efficiency stems from sifting out unpopular objects over time. Sieve transcends a single standalone algorithm — it can also be embedded within other cache policies to design more advanced algorithms.

This chapter makes the following contributions.

- It presents a simple, fast, and surprisingly efficient cache eviction algorithm, SIEVE, for web caches.
- It demonstrates Sieve's simplicity by implementing it in five production cache libraries by changing less than 20 lines of code on average.
- It shows that Sieve outperforms all state-of-the-art eviction algorithms on more than 45% of a large dataset of 1559 traces.
- It illustrates Sieve's scalability using Cachelib-based implementation, which achieves 17% and 125% higher throughput than optimized LRU at 1 and 16 threads.
- It shows how Sieve, as a turn-key cache primitive, opens new opportunities for designing advanced eviction algorithms, e.g., replacing the LRU in ARC, TwoQ, and LeCaR with Sieve.

Part IV

Wrapping Up

Chapter 10

Lessons learned and looking forward

10.1 Lessons learned about industry adoption

Several of the works discussed in this thesis, i.e., Segcache and S3-FIFO, have been deployed in real systems. However, some other works have not, e.g., GL-Cache and C2DN. When looking back on what helps the community adopt the projects, an important factor is the intensity and criticality of the pain point being solved. The large-scale measurements that I have performed have been instrumental to the designs. For example, Segcache (Chapter 4) leverages the insights from the workload analysis (Chapter 3) and prioritizes TTL expiration and small object metadata overhead. Moreover, S3-FIFO (Chapter 8) leverages the insights from the eviction algorithm measurement (Chapter 7), which shows that quickly evicting new objects is important. While measurement is critical, in essence, it is crucial to choose the right problem to work on.

Another interesting observation on adoption is that a new system design is often very hard for the community to adopt. This is mainly because new system design often requires significant re-implementation or sometimes starting from scratch. The risk of running less mature software often outweighs the efficiency gain. As a comparison, new algorithms are often easier to adopt because they often require smaller changes to existing systems.

10.2 Future work: Next-gen cache management systems

The works presented in this thesis demonstrated that we can improve both space management and cache replacement of a cache management system to achieve high efficiency. With these new designs, we can achieve high efficiency and scalability with simple design, but is this the end?

10.2.1 Better cache replacement with machine learning

Chapter 7, Chapter 8, and Chapter 9 demonstrate that simple primitives are sufficient to design efficient and scalable cache eviction algorithms that outperform state-of-the-art designs. However, they use static queues with magic parameters, which I found suboptimal for some workloads, can we use machine learning to help with choosing the parameters?

Chapter 6 shows a co-design of storage and eviction — using group-level learning on segment-structured caches. Although the amortization allows GL-Cache to have a lower storage and computation overhead compared to traditional object-level learning, performing inference at runtime, i.e., at eviction time, is not ideal for production because of two reasons. First, the amount of computation needed by inference scales with the request rate and miss ratio, which may lead to a death spiral. Request rate and miss ratio increase often correlate with the increase of system load; spending more CPU cycles on training and inference can potentially lead to cascading failure. Second, caching is a lightweight process, even the computation of simple model inference, e.g., gradient-boosting tree, is a significant overhead. In most production systems, such overhead is not acceptable.

Besides the overhead problem in existing learned caches, there are two other issues — interpretability and robustness. Having the ability to explain the decision of a learned cache is important for critical infrastructure components such as caching systems because the failure of caching systems often leads to cascading failure [129, 216]. However, existing learned caches, including LRB [312] and GL-Cache, cannot be explained or reasoned about. First, these learned caches all use the machine learning model as a black box, the decisions of which cannot be interpreted. Second, existing learned caches learn the reuse distances of individual objects, which are inherently hard to predict. Moreover, the metric we care about, miss ratio, is the aggregated results of many evictions and has no relationship with the prediction of an individual object. For example, a learned cache predicting when an object will be requested in the future may use mean squared error as the loss function, which minimizes the predicted reuse distance and true reuse distance. However, two models achieving the same miss squared error often do not provide the same miss ratio. Therefore, future work can design more interpretable learned caches that optimize for the end-to-end miss ratio.

Besides interpretability, it is also important for the learned cache to be robust. Existing learned caches all require frequent retraining because they learn short-term (local) access patterns. For example, the reuse distance of an object during day and night can be significantly different, which dictates the need for frequent retraining. However, learning short access patterns not only poses a huge overhead but also renders the learned cache less robust — a small fluctuation in access pattern can cause unnecessary adaptations that hurt the miss ratio. Therefore, future work can look into the design of learned caches with robustness. One possible direction would be learning high-level patterns and predicting high-level parameters, such as the queue size in S3-FIFO. In addition, although multiple works claimed that adapting to workload changes is important [291, 312], it is unclear whether real-time adaptations are necessary or even helpful in caching. I conjecture that for most large production systems, the workloads are often stable enough that they do not require quick adaptations in real time. Therefore, we only need to periodically learn the parameters.

10.2.2 Caching on flash

DRAM is both more expensive and also emits significantly more embodied and operational carbon per GB compared with SSDs [337]. For the sustainability of our society, it is increasingly important to move some of the DRAM usage to SSD, especially considering that SSDs today can often provide high bandwidth in the same order as a single DRAM DIMM.

Several cache system and algorithm designs discussed in this thesis pave the way towards better flash-based caches. For example, Chapter 4 describes Segcache, a segment-structured key-value cache. Its design of segments avoids overwrites (in-place updates) and transforms small object writes to large segment writes. This benefits most of the external storage devices, such as persistent memory, flash and spinning disk. In addition, S3-FIFO uses a FIFO-only solution for caching, which is also suitable for flash caches because all FIFO writes are sequential.

Existing works focus on a specific workload type, e.g., small objects [225], photo distribution [320]. Future works can look into both more general multi-tenanted caching workloads as well as more specific application workloads and design better flash caches with high performance and endurance.

10.3 The missing components of caching

10.3.1 Cache consistency

"There are two hard problems in computer science: naming things, cache invalidation, and off-by-one error."

–Phil Karlton

This thesis focuses on the efficiency and scalability of software caches. However, another hard problem in caching is consistency. Because caches are increasingly deployed as a distributed cluster, inconsistency between the cache and the backend is not rare. Moreover, invalidating derived data is often non-trivial. For example, some computed results may need to be invalidated when the underlying data used for computation has changed.

In key-value caches, Time-to-live (TTL) is often used to guarantee that stale data does not stay forever. While effective, TTL brings a new challenge — how to set the value so that we can minimize the stale data without significantly increasing the miss ratio. Moreover, it would also be interesting to explore whether we can design better approaches to find all the data that need invalidation and achieve consistency without sacrificing miss ratio.

10.3.2 Cache security

While consistency, efficiency, and scalability are critical for a sustainable society, several other aspects of caching are also crucial for our infrastructure in the digital age. For example, the security of a software cache. There have been many reports of cache deception attacks [72, 73] and cache poisoning attacks [134], causing sensitive information to be leaked or serving wrong content to the victim. There have been many research works on hardware cache security, e.g., side-channel attacks; however, the study of software cache security is scarce and should be one of the future focuses.

10.4 Concluding Remarks

As the world continues to depend on software caches for fast and efficient data access, the deployment of software caches will keep expanding. However, the manufacturing and operation of cache servers result in significant carbon emissions. To create a more sustainable society, it is crucial to enhance cache efficiency and scalability to reduce DRAM and CPU consumption.

Bibliography

[1] How to interpret r-squared and goodness-of-fit in regression analysis. https://www.datasciencecentral.com/profiles/blogs/regression-analysis-how-do-i-interpret-r-squared-and-assess-the, 2023. Accessed: 2023-05-28. 3.3.5

[2] 0. Alibaba block-trace. https://github.com/alibaba/block-traces, 2024. Accessed: 2024-07-20. 7.1

[3] 0. Running cachebench with the trace workload. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval, 2024. Accessed: 2024-07-20. 8.1, 9.3.1, 9.1

[4] Atul Adya, John Dunagan, and Alec Wolman. Centrifuge: Integrated Lease Management and Partitioning for Cloud Services. In *7th USENIX Symposium on Networked Systems Design and Implementation (NSDI 10)*, NSDI'10, 2010. URL https://www.usenix.org/conference/nsdi10-0/centrifuge-integrated-lease-management-and-partitioning-cloud-services. 5.7

[5] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, and others. Slicer:Auto-Sharding for Datacenter Applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, 2016. 5.7

[6] Vaneet Aggarwal, Yih-Farn Robin Chen, Tian Lan, and Yu Xiang. Sprout: A Functional Caching Approach to Minimize Service Latency in Erasure-Coded Storage. *IEEE/ACM Transactions on Networking*, 25(6):3683–3694, December 2017. ISSN 1558-2566. doi: 10.1109/TNET.2017.2749879. URL https://ieeexplore.ieee.org/document/8048491. Conference Name: IEEE/ACM Transactions on Networking. 5.7

[7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design Tradeoffs for SSD Performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 57–70, USA, 2008. USENIX Association. URL https://www-usenix-org.cmu.idm.oclc.org/legacy/events/usenix08/tech/full_papers/agrawal/agrawal.pdf. 8.1.1

[8] M.K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *2005 International Conference on Dependable Systems and Networks (DSN'05)*, pages 336–345, June 2005. doi: 10.1109/DSN.20

05.96. URL https://ieeexplore.ieee.org/document/1467808. ISSN: 2158-3927. 5.7

[9] Alfred V Aho, Peter J Denning, and Jeffrey D Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971. 5.3.3

[10] Mehmet Altinel, Christof Bornhoevd, Chandrasekaran Mohan, Mir Hamid Pirahesh, Berthold Reinwald, and Saileshwar Krishnamurthy. System and method for adaptive database caching, July 1 2008. US Patent 7,395,258. 3.1.4

[11] AMD. Amd server processor specifications. https://www.amd.com/en/products/specifications/server-processor.html, 2024. Accessed: 2024-12-06. 1.1, 2.2, 8.1.1

[12] Chris Aniszczyk. Caching with twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html, 2024. Accessed: 2024-07-20. 4.3.2

[13] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 129–142, USA, May 2005. USENIX Association. 5.7

[14] Apache. Apache traffic server. https://trafficserver.apache.org/, 2024. Accessed: 2024-07-20. 2.3.2, 3.5.1, 5.3.3, 5.5.7, 5.8, 8.1.1, 8.4.4

[15] Apache. Apache aurora. http://aurora.apache.org/, 2024. Accessed: 2024-07-20. 3.1.1

[16] Apache. Apache mesos. http://mesos.apache.org/, 2024. Accessed: 2024-07-20. 3.1.1

[17] ARC. Adaptive replacement cache implementation in python. https://gist.github.com/pior/da3b6268c40fa30c222f, 2024. Accessed: 2024-07-20. 8.4.1

[18] Ismail Ari, Ahmed Amer, Robert B Gramacy, Ethan L Miller, Scott A Brandt, and Darrell DE Long. ACME: Adaptive Caching Using Multiple Experts. In *WDAS*, volume 2, pages 143–158, 2002. 6.1.1, 9.1.2

[19] Jose A Arjona-Medina, Michael Gillhofer, Michael Widrich, Thomas Unterthiner, Johannes Brandstetter, and Sepp Hochreiter. Rudder: Return decomposition for delayed rewards. *Advances in Neural Information Processing Systems*, 32, 2019. 6.1.1

[20] M. Arlitt and T. Jin. A workload characterization study of the 1998 World Cup Web site. *IEEE Network*, 14(3):30–37, June 2000. ISSN 08908044. doi: 10.1109/65.844498. URL http://ieeexplore.ieee.org/document/844498/. 3.7

[21] Martin Arlitt, Rich Friedrich, and Tai Jin. Workload characterization of a Web proxy in a cable modem environment. *SIGMETRICS Perform. Eval. Rev.*, 27(2): 25–36, September 1999. ISSN 0163-5999. doi: 10.1145/332944.332951. URL https://doi.org/10.1145/332944.332951. 3.7

[22] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evalu-

ating content management techniques for Web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, March 2000. ISSN 0163-5999. doi: 10.1145/346000.346003. 2.3.1, 4.2.6

[23] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance Evaluation of Web Proxy Cache Replacement Policies. In *Perform. Eval.*, volume 39, pages 149–164, NLD, February 2000. Elsevier Science Publishers B. V. doi: 10.1016/S0166-5316(99)00062-0. 4.2.6, 4.5.1, 6.4

[24] M.F. Arlitt and C.L. Williamson. Internet Web servers: workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, October 1997. ISSN 10636692. doi: 10.1109/90.649565. URL `http://ieeexplore.ieee.org/document/649565/`. 3.7

[25] 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. Association for Computing Machinery. ISBN 978-1-4503-1097-0. doi: 10.1145/2254756.2254766. 1.2, 3.3.2, 3.3.6, 3.6.1, 3.6.4, 3.7, 3.8, 4.1.2, 4.3.2, 6.1.1, 7.4, 9.3.2, 9.3.3

[26] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S. Berger. Caching with Delayed Hits. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM'20, pages 495–513, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi: 10.1145/3387514.3405883. 8.1, 9.1.1

[27] Tahir Azim, Manos Karpathiotakis, and Anastasia Ailamaki. ReCache: reactive caching for fast analytics over heterogeneous data. *Proceedings of the VLDB Endowment*, 11(3):324–337, November 2017. ISSN 2150-8097. doi: 10.14778/3157794.3157801. 2.3.1

[28] Sung Hoon Baek and Kyu Ho Park. Prefetching with adaptive cache culling for striped disk arrays. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008. 8.6

[29] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a distributed file system. *SIGOPS Oper. Syst. Rev.*, 25(5):198–212, September 1991. ISSN 0163-5980. doi: 10.1145/121133.121164. URL `https://doi.org/10.1145/121133.121164`. 3.7

[30] Mahesh Balakrishnan, Asim Kadav, Vijayan Prabhakaran, and Dahlia Malkhi. Differential RAID: Rethinking RAID for SSD reliability. *ACM Trans. Storage*, 6(2):4:1–4:22, July 2010. ISSN 1553-3077. doi: 10.1145/1807060.1807061. URL `https://doi.org/10.1145/1807060.1807061`. 5.7

[31] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. SILK: Preventing Latency Spikes in Log-Structured

Merge Key-Value Stores. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, ATC'19, pages 753–766, 2019. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/balmau. 3.8

[32] Jiwoo Bang, Chungyong Kim, Sunggon Kim, Qichen Chen, Cheongjun Lee, Eun-Kyu Byun, Jaehwan Lee, and Hyeonsang Eom. Finer-LRU: A Scalable Page Management Scheme for HPC Manycore Architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IPDPS, pages 567–576, May 2021. doi: 10.1109/IPDPS49936.2021.00065. ISSN: 1530-2075. 7.1

[33] Sorav Bansal and Dharmendra S. Modha. CAR: Clock with Adaptive Replacement. In *3rd USENIX Conference on File and Storage Technologies*, FAST'04, 2004. 6.4, 7.1, 7.3, 8.6, 9.1.2

[34] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. Adaptive TTL-Based Caching for Content Delivery. In *Proceedings of the 2017 ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'17, pages 45–46, Urbana-Champaign Illinois USA, June 2017. ACM. ISBN 978-1-4503-5032-7. doi: 10.1145/3078505.3078560. URL https://dl.acm.org/doi/10.1145/3078505.3078560. 8.1, 8.2.1

[35] Alexandros Batsakis, Randal Burns, Arkady Kanevsky, James Lentini, and Thomas Talpey. AWOL: an adaptive write optimizations layer. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 1–14, USA, February 2008. USENIX Association. 8.6

[36] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture*, HPCA'15, pages 64–75, Burlingame, CA, USA, February 2015. IEEE. ISBN 978-1-4799-8930-0. doi: 10.1109/HPCA.2015.7056022. 3.1.4, 3.5.3, 4.1, 7.1, 8.6, 9.1.2

[37] Nathan Beckmann and Daniel Sanchez. Maximizing Cache Performance Under Uncertainty. In *2017 IEEE International Symposium on High Performance Computer Architecture*, HPCA'17, pages 109–120, Austin, TX, February 2017. IEEE. ISBN 978-1-5090-4985-1. doi: 10.1109/HPCA.2017.43. 9.1.2

[38] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX symposium on networked systems design and implementation*, NSDI'18, pages 389–403, 2018. 1.1, 1.3, 2.3.1, 3.1.4, 3.3.4, 3.3.4, 3.6.4, 3.8, 4.1, 4.1.1, 4.1.4, 4.2.6, 4.3.2, 4.3.4, 4.5.1, 6.1.1, 6.1.1, 6.3.1, 7.4, 7.4, 7.4, 7.5, 8.2.2, 8.4.2, 8.6, 9.1.2, 9.1.3, 9.2.2, 9.4.1

[39] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. ISSN 0018-8670. doi: 10.1147/sj.52.0078. 6.1.1, 6.2.4, 7.1, 7.4, 9.1.2

[40] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of the ACM*, 12(6):349–353, June 1969. ISSN 0001-0782. doi: 10.1145/363011.363155. 8.5.2

[41] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 753–768. USENIX Association, November 2020. ISBN 978-1-939133-19-9. 3.1.1, 3.1.4, 3.6.3, 4.1.1, 4.4.2, 5.1, 5.2.4, 5.5.7, 6.1.1, 6.3.1, 6.4, 7.2, 7.4, 7.5, 8.1.1, 8.1.2, 8.4.4, 9.1.1, 9.2.2

[42] Daniel S. Berger. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, Hotnets'18, pages 134–140, Redmond WA USA, November 2018. ACM. ISBN 978-1-4503-6120-0. doi: 10.1145/3286062.3286082. 1.1, 4.1, 6.1.1, 6.1.1, 6.2.5, 6.3.1, 7.1

[43] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. AdaptSize: Orchestrating the hot object memory cache in a content delivery network. In *14th USENIX symposium on networked systems design and implementation*, NSDI'17, pages 483–498, 2017. 1.1, 2.3.1, 4.1, 4.5.1, 5.2.1, 6.4, 7.1, 7.5, 8.1, 8.4.4, 8.5.1

[44] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. Practical Bounds on Optimal Caching with Variable Object Sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–38, June 2018. ISSN 2476-1249. doi: 10.1145/3224427. 6.2.4, 6.3.1, 6.3.2

[45] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. RobinHood: Tail latency aware caching – dynamic reallocation from Cache-Rich to Cache-Poor. In *13th USENIX symposium on operating systems design and implementation*, OSDI'18, pages 195–212, Carlsbad, CA, October 2018. USENIX Association. ISBN 978-1-939133-08-3. 7.1

[46] Adit Bhardwaj and Vaishnav Janardhan. Pecc: Prediction-error correcting cache. In *Workshop on ML for Systems at NeurIPS*, volume 32, 2018. 6.1.1, 9.1.2

[47] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 359–374, Santa Clara, CA, February 2017. USENIX Association. ISBN 978-1-931971-36-2. URL https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling. 5.2.4

[48] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX annual technical conference*, ATC'17, pages 499–511, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. 2.3.1, 3.1.4, 3.3.4, 3.3.4, 3.6.4, 3.8, 4.1, 4.1.1, 4.1.4, 4.2.6, 4.3.2, 4.5.1, 6.3.5, 6.4, 7.4, 7.5, 8.6, 9.2.2

[49] Simona Boboila and Peter Desnoyers. Write Endurance in Flash Drives: Measurements and Analysis. In *Proceedings of the 8th USENIX conference on File and stroage technologies*, FAST'10, pages 115–128, 2010. 8.1.1

[50] Bradfitz. group cache. https://github.com/golang/groupcache, 2024.

Accessed: 2024-07-20. 8.1.2, 9.2.2, 9.2

[51] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and Zipf-like distributions: evidence and implications. In *Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 126–134 vol.1, New York, NY, USA, 1999. IEEE. ISBN 978-0-7803-5417-3. doi: 10.1109/INFCOM.1999.749260. 7.4, 9.1.1

[52] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. On the implications of Zipf's law for web caching. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1998. 9.1.1, 9.4.3

[53] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999. 3.3.5, 3, 3.6.3, 6.2.2

[54] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, ATC'13, pages 49–60, 2013. ISBN 978-1-931971-01-0. 2.1, 3.1.1

[55] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS'21, pages 221–227, New York, NY, USA, June 2021. Association for Computing Machinery. ISBN 978-1-4503-8438-4. doi: 10.1145/3458336.3465286. URL `https://dl.acm.org/doi/10.1145/3458336.3465286`. 6.2.5

[56] Ali R. Butt, Chris Gniady, and Y. Charlie Hu. The performance impact of kernel prefetching on buffer cache replacement algorithms. In *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS'05, pages 157–168, New York, NY, USA, June 2005. Association for Computing Machinery. ISBN 978-1-59593-022-4. doi: 10.1145/1064212.1064231. 8.6

[57] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM'02, pages 47–60, New York, NY, USA, August 2002. Association for Computing Machinery. ISBN 978-1-58113-570-1. doi: 10.1145/633025.633031. URL `https://dl.acm.org/doi/10.1145/633025.633031`. 8.1

[58] John Byers, Jeffrey Considine, and Michael Mitzenmacher. Simple Load Balancing for Distributed Hash Tables. In M. Frans Kaashoek and Ion Stoica, editors, *Peer-to-Peer Systems II*, pages 80–87, Berlin, Heidelberg, 2003. Springer. ISBN 978-3-540-

45172-3. doi: 10.1007/978-3-540-45172-3_7. 5.7

[59] Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: cloud computing for everyone. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, SOCC'11, pages 1–14, Cascais, Portugal, 2011. ACM Press. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038932. URL http://dl.acm.org/citation.cfm?doid=2038916.2038932. 5.7

[60] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mPart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM'18, pages 84–95, Philadelphia PA USA, June 2018. ACM. ISBN 978-1-4503-5801-9. doi: 10.1145/3210563.3210571. 4.5.1

[61] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS'19, pages 353–362, Washington District of Columbia USA, September 2019. ACM. ISBN 978-1-4503-7206-0. doi: 10.1145/3357526.3357562. URL https://dl.acm.org/doi/10.1145/3357526.3357562. 3.6.5, 4.1.3, 4.5.1, 4.5.2, 8.3.2

[62] cacheus. Cacheus implementation. https://github.com/sylab/cacheus, 2024. Accessed: 2024-07-20. 8.4.1

[63] caffeine. Caffeine java cache package. https://github.com/ben-manes/caffeine, 2024. Accessed: 2024-07-20. 8.4.1

[64] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX Symposium on Internet Technologies and Systems*, USITS'97, Monterey, CA, December 1997. USENIX Association. 2.3.1, 7.1, 9.1.2

[65] Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, November 1996. ISSN 0734-2071. doi: 10.1145/235543.235544. 8.6

[66] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H.C. Du. Characterizing, modeling, and benchmarking RocksDB Key-Value workloads at facebook. In *18th USENIX conference on file and storage technologies*, FAST'20, pages 209–223, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. 3.6.4, 3.7, 6.1.1

[67] Richard W. Carr and John L. Hennessy. WSCLOCK: a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP'81, pages 87–95, New York, NY, USA, December 1981. Association for Computing Machinery. ISBN 978-0-89791-062-0. doi: 10.1145/800216.806596. 9.4.1

[68] Dirk G Cattrysse and Luk N Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European journal of operational research*, 60(3): 260–272, 1992. 5.3.2

[69] Yunpeng Chai, Zhihui Du, Xiao Qin, and David A. Bader. WEC: Improving Durability of SSD Cache Drives by Caching Write-Efficient Data. *IEEE Transactions on Computers*, 64(11):3304–3316, November 2015. ISSN 1557-9956. doi: 10.1109/TC.2015.2401029. Conference Name: IEEE Transactions on Computers. 8.1.1, 8.6

[70] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD/PODS'18, pages 275–290, Houston TX USA, May 2018. ACM. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196898. URL https://dl.acm.org/doi/10.1145/3183713.3196898. 4.1.2, 4.2.3, 7.1, 7.2, 8.3.2

[71] Zheng Chang, Lei Lei, Zhenyu Zhou, Shiwen Mao, and Tapani Ristaniemi. Learn to Cache: Machine Learning for Network Edge Caching in the Big Data Era. *IEEE Wireless Communications*, 25(3):28–35, June 2018. ISSN 1536-1284, 1558-0687. doi: 10.1109/MWC.2018.1700317. 6.1.1

[72] chatgpt. Chatgpt account takeover - wildcard web cache deception. https://nokline.github.io/bugbounty/2024/02/04/ChatGPT-ATO.html, 2024. Accessed: 2024-07-28. 10.3.2

[73] chatgpt. Shockwave identifies web cache deception and account takeover vulnerability affecting openai's chatgpt. https://www.shockwave.cloud/blog/shockwave-works-with-openai-to-fix-critical-chatgpt-vulnerability, 2024. Accessed: 2024-07-28. 10.3.2

[74] H. Che, Z. Wang, and Y. Tung. Analysis and design of hierarchical Web caching systems. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1416–1424, Anchorage, AK, USA, 2001. IEEE. ISBN 978-0-7803-7016-6. doi: 10.1109/INFCOM.2001.916637. URL http://ieeexplore.ieee.org/document/916637/. 8.2.1, 9.4.2

[75] Fangfei Chen, Ramesh K. Sitaraman, and Marcelo Torres. End-User Mapping: Next Generation Request Routing for Content Delivery. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM'15, pages 167–181, London United Kingdom, August 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787500. URL https://dl.acm.org/doi/10.1145/2785956.2787500. 5.1

[76] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies*, FAST'20, pages 239–252, 2020. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/chen-jiqiang. 3.1.4, 3.3.2, 3.3.5

[77] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural*

Support for Programming Languages and Operating Systems, ASPLOS'20, pages 1077–1091, Lausanne Switzerland, March 2020. ACM. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378515. URL https://dl.acm.org/doi/10.1145/3373376.3378515. 4.5.2

[78] Zhifeng Chen, Yuanyuan Zhou, and Kai Li. Eviction-based Cache Placement for Storage Caches. In *USENIX Annual Technical Conference, General Track*, ATC'03, pages 269–281, 2003. 8.6

[79] Liangfeng Cheng, Yuchong Hu, and Patrick P. C. Lee. Coupling Decentralized Key-Value Stores with Erasure Coding. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'19, pages 377–389, Santa Cruz CA USA, November 2019. ACM. ISBN 978-1-4503-6973-2. doi: 10.1145/3357223.3362713. URL https://dl.acm.org/doi/10.1145/3357223.3362713. 5.7

[80] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1–16, New York, NY, USA, April 2015. Association for Computing Machinery. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741967. 3.1.4

[81] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, ATC'20, pages 379–392, 2016. 8.6

[82] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Citeseer, 1998. 4.2.6, 4.2.6, 4.5.1, 6.4

[83] Boris V Cherkassky and Andrew V Goldberg. On implementing the push—relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997. 5.4

[84] Jongmoo Choi, Sam H Noh, Sang Lyul Min, and Yookun Cho. An implementation study of a detection-based adaptive block replacement scheme. In *USENIX annual technical conference*, ATC'99, pages 239–252, 1999. 8.6

[85] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX workshop on hot topics in cloud computing*, HotCloud'15, 2015. 3.6.5, 8.6

[86] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX symposium on networked systems design and implementation*, NSDI'16, pages 379–392, 2016. 3.1.4, 3.6.5, 3.8, 4.1, 8.6, 9.1.2

[87] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX annual technical conference*, ATC'17, pages 321–334, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. 3.1.4, 3.8, 4.1, 4.1.3, 4.2.2, 4.5.2, 7.1, 8.6

[88] CISCO. Global IP traffic forecast: The zettabyte era—trends and analysis, June

2017. Available at https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf, accessed 24/09/17. 5.8

[89] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010. 3.6.3, 3.8, 9.1.1

[90] Fernando J Corbato. A paging experiment with the multics system. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE PROJECT MAC, 1968. 7.1, 8.1.2, 9.1.2, 9.2.2, 9.4.1

[91] Michael D Dahlin, Randolph Y Wang, Thomas E Anderson, and David A Patterson. Cooperative caching: Using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, OSDI'94, pages 19–es, 1994. 7.1, 8.6

[92] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968. 7.1, 9.1.2

[93] Peter J. Denning. Working Set Analytics. *ACM Computing Surveys*, 53(6):1–36, November 2021. ISSN 0360-0300, 1557-7341. doi: 10.1145/3399709. 7.1, 7.3

[94] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. Garbage-first garbage collection. In *Proceedings of the 4th international symposium on Memory management*, pages 37–48, 2004. 7.5

[95] Meta developers. Cachelib - pluggable caching engine to build and scale high performance cache services. https://cachelib.org/, 2024. Accessed: 2024-07-20. 1.1, 8.4.1, 9.1.1, 9.7

[96] Meta developers. Distributed mutex. https://github.com/facebook/folly/blob/2c00d14adb9b632936f3abfbf741373871cd64a6/folly/synchronization/DistributedMutex.h, 2024. Accessed: 2024-07-20. 9.3.3

[97] Pelikan developers. Pelikan. https://github.com/pelikan-io/pelikan, 2024. Accessed: 2024-07-20. 6.3.1, 8.1.2, 9.1.1

[98] Diego Didona and Willy Zwaenepoel. Size-aware Sharding For Improving Tail Latencies in In-memory Key-value Stores. In *16th USENIX Symposium on Networked Systems Design and Implementation*, NSDI19, pages 79–94, 2019. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/didona. 3.1.4, 3.3.5

[99] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6 (5):50–58, 2002. 5.1

[100] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Man-*

agement of Data, SIGMOD/PODS'20, pages 969–984, Portland OR USA, June 2020. ACM. ISBN 978-1-4503-6735-6. doi: 10.1145/3318464.3389711. URL https://dl.acm.org/doi/10.1145/3318464.3389711. 6.4

[101] Jialin Ding, Ryan Marcus, Andreas Kipf, Vikram Nathan, Aniruddha Nrusimha, Kapil Vaidya, Alexander van Renen, and Tim Kraska. SageDB: An Instance-Optimized Data Analytics System. *Proceedings of the VLDB Endowment*, 15(13): 4062–4078, September 2022. ISSN 2150-8097. doi: 10.14778/3565838.3565857. URL https://dl.acm.org/doi/10.14778/3565838.3565857. 6.4

[102] Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *USENIX Annual Technical Conference*, volume 7 of *ATC'07*, pages 261–274, 2007. 8.6

[103] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012. 6.2.5

[104] Siying Dong. Reducing lock contention in rocksdb. https://rocksdb.org/blog/2014/05/14/lock.html, 2024. Accessed: 2024-12-06. 8.1.2

[105] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, S.H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: a spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, December 2001. ISSN 0018-9340. doi: 10.1109/TC.2001.970573. 1.3, 2.3.1, 6.4, 7.1, 9.1.2

[106] Rémi Dulong, Rafael Pires, Andreia Correia, Valerio Schiavoni, Pedro Ramalhete, Pascal Felber, and Gaël Thomas. NVCache: A Plug-and-Play NVMM-based I/O Booster for Legacy Systems. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 186–198, June 2021. doi: 10.1109/DSN48987.2021.00033. URL http://arxiv.org/abs/2105.10397. arXiv:2105.10397 [cs]. 8.1

[107] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019. URL https://www.flux.utah.edu/paper/duplyakin-atc19. 6.3.1, 8.4.1, 9.3.1

[108] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy, December 2015. 2.3.1, 3.1.4, 4.2.6, 4.5.1, 6.4, 7.5

[109] Gil Einziger, Roy Friedman, and Ben Manes. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage*, 13(4):1–31, December 2017. ISSN 1553-3077, 1553-3093. doi: 10.1145/3149371. 1.1, 1.3, 2.3.1, 4.1, 4.5.1, 6.3.1, 6.4, 7.5, 8.1.2, 8.4.2, 9.1.2

[110] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive Software Cache Management. In *Proceedings of the 19th International Middleware Conference*,

Middleware'18, pages 94–106, Rennes France, November 2018. ACM. ISBN 978-1-4503-5702-9. doi: 10.1145/3274808.3274816. 6.4, 7.5, 9.1.2

[111] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. Lightweight robust size aware cache management. *ACM Transactions on Storage*, 18(3), August 2022. ISSN 1553-3077. doi: 10.1145/3507920. 6.4, 9.1.2

[112] Daniel E. Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 523–535, Santa Clara, CA, March 2016. USENIX Association. ISBN 978-1-931971-29-4. URL `https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud`. 5.7

[113] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX symposium on networked systems design and implementation*, NSDI'19, pages 65–78, Boston, MA, February 2019. USENIX Association. ISBN 978-1-931971-49-2. 3.1.4, 4.1, 4.5.1, 5.1, 5.5.7, 5.7, 6.1.1, 7.5, 8.1.1, 8.6, 8.7, 9.1.2

[114] Assaf Eisenman, Maxim Naumov, Darryl Gardner, Misha Smelyanskiy, Sergey Pupyrev, Kim Hazelwood, Asaf Cidon, and Sachin Katti. Bandana: Using non-volatile memory for storing deep learning models. In A. Talwalkar, V. Smith, and M. Zaharia, editors, *Proceedings of machine learning and systems*, volume 1 of *mlsys'20*, pages 40–52, 2019. 7.1, 9.1.2

[115] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. It's time to revisit LRU vs. FIFO. In *12th USENIX workshop on hot topics in storage and file systems*, hotStorage'20. USENIX Association, July 2020. 7.1, 7.2, 8.1, 9.1.2

[116] Ronald Fagin. Asymptotic miss ratios over independent references. *Journal of Computer and System Sciences*, 14(2):222–250, 1977. 5.3.3, 5.3.3

[117] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. Small cache, big effect: provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, SOCC'11, pages 1–12, Cascais, Portugal, 2011. ACM Press. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038939. 3.3.5, 3.6.3, 5.7, 8.1, 8.6

[118] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX symposium on networked systems design and implementation*, NSDI'13, pages 371–384, 2013. 1.1, 3.1.4, 3.4.2, 3.8, 4.1, 4.1.2, 4.1.4, 4.2.3, 4.5.1, 4.5.2, 6.4, 7.1, 7.2, 7.3, 8.4.3, 8.6, 9.1.2

[119] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. PA-Cache: Evolving Learning-Based Popularity-Aware Content Caching in Edge Networks, December 2020. 6.1.1, 8.1

[120] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. Feedforward Neural Networks for Caching: n Enough or Too Much? *ACM SIGMETRICS Performance Evaluation Review*, 46(3):139–142, January 2019. ISSN 0163-5999. doi: 10.1145/3308897.3308958. 6.1.1

[121] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'22, pages 395–411, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-28-1. 7.1, 8.1, 8.1.2, 8.6

[122] Philippe Flajolet, Daniele Gardy, and Loÿs Thimonier. Birthday paradox, coupon collectors, caching algorithms and self-organizing search. *Discrete Applied Mathematics*, 39(3):207–229, 1992. 9.4.2

[123] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010. 5.2.2

[124] Christine Fricker, Philippe Robert, and James Roberts. A versatile and accurate approximation for LRU cache performance. In *ITC*, page 8, 2012. 5.3.3

[125] Alexander Fuerst and Prateek Sharma. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'21, pages 386–400, Virtual USA, April 2021. ACM. ISBN 978-1-4503-8317-2. doi: 10.1145/3445814.3446757. 7.1

[126] Rohan Gandhi, Hongqiang Harry Liu, Y. Charlie Hu, Guohan Lu, Jitendra Padhye, Lihua Yuan, and Ming Zhang. Duet: cloud scale load balancing with hardware and software. In *Proceedings of the 2014 ACM conference on SIGCOMM*, SIGCOMM '14, pages 27–38, New York, NY, USA, August 2014. Association for Computing Machinery. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626317. URL https://doi.org/10.1145/2619239.2626317. 5.7

[127] Jim Gao. Machine learning applications for data center optimization. https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/42542.pdf, 2014. Accessed: 2022-04-06. 6.4

[128] gdpr. Art. 17 gdpr right to erasure ('right to be forgotten'). https://gdpr-info.eu/art-17-gdpr/, 2023. Accessed: 2023-02-06. 3.3.4, 4.1.1

[129] Geeks for geeks. How a cache stampede caused one of facebook's biggest outages. https://www.geeksforgeeks.org/gfact-how-a-cache-stampede-caused-one-of-facebooks-biggest-outages, 2023. Accessed: 2024-07-28. 10.2.1

[130] Binny S Gill. On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions. In *FAST*, volume 8 of *FAST'08*, pages 1–17,

2008. 8.6

[131] Binny S Gill and Luis Angel D Bathen. AMP: Adaptive multi-stream prefetching in a shared cache. In *FAST*, volume 7, pages 185–198, 2007. 8.6

[132] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC'07, pages 15–28, New York, NY, USA, October 2007. Association for Computing Machinery. ISBN 978-1-59593-908-1. doi: 10.1145/1298306.1298310. URL https://doi.org/10.1145/1298306.1298310. 9.1.1

[133] github. Memory used only grows despite unlink/delete of keys. https://github.com/redis/redis/issues/7482, 2024. Accessed: 2024-07-20. 4.1.1, 4.4.2

[134] glassdoor. Caching the un-cacheables - abusing url parser confusions (web cache poisoning technique). https://nokline.github.io/bugbounty/2022/09/02/Glassdoor-Cache-Poisoning.html, 2024. Accessed: 2024-07-28. 10.3.2

[135] Wolfram Gloger. ptmalloc. http://www.malloc.de/en/, 2024. Accessed: 2024-07-20. 3.1.3

[136] Robert B. Gramacy, Manfred K. Warmuth, Scott A. Brandt, and Ismail Ari. Adaptive caching by refetching. In *Proceedings of the 15th International Conference on Neural Information Processing Systems*, NIPS'02, pages 1489–1496, Cambridge, MA, USA, 2002. MIT Press. 6.1.1

[137] Xiaoming Gu and Chen Ding. On the theory and potential of LRU-MRU collaborative cache management. In *Proceedings of the international symposium on Memory management*, ISMM '11, pages 43–54, New York, NY, USA, June 2011. Association for Computing Machinery. ISBN 978-1-4503-0263-0. doi: 10.1145/1993478.1993485. URL https://doi.org/10.1145/1993478.1993485. 9.1.2

[138] Yu Guan, Xinggong Zhang, and Zongming Guo. CACA: Learning-based Content-aware Cache Admission for Video Content in Edge Caching. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM'19, pages 456–464, New York, NY, USA, October 2019. Association for Computing Machinery. ISBN 978-1-4503-6889-6. doi: 10.1145/3343031.3350890. 6.4

[139] Ajay Gulati, Chethan Kumar, Irfan Ahmad, and Karan Kumar. BASIL: Automated IO Load Balancing Across Storage Devices. In *8th USENIX Conference on File and Storage Technologies (FAST 10)*, FAST'10, 2010. URL https://www.usenix.org/conference/fast-10/basil-automated-io-load-balancing-across-storage-devices. 5.7

[140] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. The stretched exponential distribution of internet media access patterns. In *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, pages 283–294, New York, NY, USA, August 2008. Association for Computing Machinery. ISBN 978-1-59593-989-0. doi: 10.1145/1400751.1400789. URL https:

//doi.org/10.1145/1400751.1400789. 9.1.1

[141] Raluca Halalai, Pascal Felber, Anne-Marie Kermarrec, and François Taïani. Agar: A Caching System for Erasure-Coded Data. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 23–33, June 2017. doi: 10.1109/ICDCS.2017.97. ISSN: 1063-6927. 5.7

[142] Beining Han, Zhizhou Ren, Zuofan Wu, Yuan Zhou, and Jian Peng. Off-policy reinforcement learning with delayed rewards, 2021. URL https://arxiv.org/abs/2106.11854. 6.1.1

[143] Mingzhe Hao, Huaicheng Li, Michael Hao Tong, Chrisma Pakha, Riza O. Suminto, Cesar A. Stuardo, Andrew A. Chien, and Haryadi S. Gunawi. MittOS: Supporting Millisecond Tail Tolerance with Fast Rejecting SLO-Aware OS Interface. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 168–183, New York, NY, USA, October 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132774. URL https://dl.acm.org/doi/10.1145/3132747.3132774. 8.1.1

[144] Mingzhe Hao, Levent Toksoz, Nanqinqin Li, Edward Edberg Halim, Henry Hoffmann, and Haryadi S Gunawi. LinnOS: Predictability on Unpredictable Flash Storage with a Light Neural Network. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2020. 8.1.1

[145] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K. Sitaraman. Trade-offs in optimizing the cache deployments of CDNs. In *IEEE INFOCOM 2014 - IEEE Conference on Computer Communications*, IEEEConferenceon Computer Communications'14, pages 460–468, Toronto, ON, Canada, April 2014. IEEE. ISBN 978-1-4799-3360-0. doi: 10.1109/INFOCOM.2014.6847969. URL http://ieeexplore.ieee.org/document/6847969/. 9.1.1

[146] Milad Hashemi, Kevin Swersky, Jamie A. Smith, Grant Ayers, Heiner Litz, Jichuan Chang, Christos Kozyrakis, and Parthasarathy Ranganathan. Learning Memory Access Patterns. In *International Conference on Machine Learning*, ICML'18, March 2018. 6.4

[147] Jeff Heaton. An empirical analysis of feature engineering for predictive modeling. In *SoutheastCon 2016*, pages 1–6. IEEE, 2016. 6.2.5

[148] HHVMLRUCache. Hhvm concurrent lru cache. https://github.com/facebook/hhvm/blob/master/hphp/util/concurrent-lru-cache.h, 2024. Accessed: 2024-07-25. 7.5

[149] V. Holmedahl, B. Smith, and Tao Yang. Cooperative caching of dynamic content on a distributed Web server. In *Proceedings. The Seventh International Symposium on High Performance Distributed Computing (Cat. No.98TB100244)*, pages 243–250, July 1998. doi: 10.1109/HPDC.1998.709978. URL https://ieeexplore.ieee.org/document/709978. ISSN: 1082-8907. 5.7

[150] Yu-Ju Hong and Mithuna Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th annual*

Symposium on Cloud Computing, SOCC'13, pages 1–17, Santa Clara California, October 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2525970. URL https://dl.acm.org/doi/10.1145/2523616.2525970. 5.7, 8.6

[151] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference*, ATC'15, pages 57–69, 2015. 3.1.3, 3.6.5, 4.1.3, 4.5.1, 6.3.3, 8.6

[152] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *2016 USENIX annual technical conference*, ATC'16, pages 351–364, Denver, CO, June 2016. USENIX Association. ISBN 978-1-931971-30-0. 9.1.2

[153] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'22, pages 73–90, 2022. ISBN 978-1-939133-28-1. URL https://www.usenix.org/conference/osdi22/presentation/huang-lexiang. 6.2.5

[154] 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, November 2013. Association for Computing Machinery. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522722. 3.1.1, 3.1.4, 3.3.1, 3.6.3, 3.7, 6.4, 7.4, 8.4.2, 9.1.1, 9.3.2, 9.4.1

[155] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. Characterizing Load Imbalance in Real-World Networked Caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets'14, pages 1–7, Los Angeles CA USA, October 2014. ACM. ISBN 978-1-4503-3256-9. doi: 10.1145/2670518.2673882. 3.3.2, 3.7, 8.6

[156] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Transactions on Storage*, 12(2):8:1–8:24, February 2016. ISSN 1553-3077. doi: 10.1145/2737832. 8.6

[157] Stratis Ioannidis and Edmund Yeh. Adaptive Caching Networks with Optimality Guarantees. In *Proceedings of the 2016 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS'16, pages 113–124, Antibes Juan-les-Pins France, June 2016. ACM. ISBN 978-1-4503-4266-7. doi: 10.1145/2896377.2901467. URL https://dl.acm.org/doi/10.1145/2896377.2901467. 9.4.2

[158] Stratis Ioannidis, Laurent Massoulie, and Augustin Chaintreau. Distributed caching over heterogeneous mobile networks. In *Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS'10, pages 311–322, 2010. 9.4.2

[159] Andhi Janapsatya, Aleksandar Ignjatovic, Jorgen Peddersen, and Sri Parameswaran. Dueling CLOCK: Adaptive cache replacement policy based on the CLOCK algorithm. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*, DATE2010, pages 920–925, Dresden, March 2010. IEEE. ISBN 978-3-9810801-6-2 978-1-4244-7054-9. doi: 10.1109/DATE.2010.5456920. URL `http://ieeexplore.ieee.org/document/5456920/`. 8.6

[160] R. Jauhari, Michael J. Carey, and Miron Livny. Priority-hints: an algorithm for priority-based buffer management. In *Proceedings of the sixteenth international conference on Very large databases*, pages 708–721, San Francisco, CA, USA, September 1990. Morgan Kaufmann Publishers Inc. 8.6

[161] jemalloc. jemalloc. `http://jemalloc.net`, 2024. Accessed: 2024-07-20. 3.1.3

[162] Yichen Jia, Zili Shao, and Feng Chen. SlimCache: Exploiting Data Compression Opportunities in Flash-Based Key-Value Caching. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, MASCOTS, pages 209–222, Milwaukee, WI, September 2018. IEEE. ISBN 978-1-5386-6886-3. doi: 10.1109/MASCOTS.2018.00029. 8.6

[163] Bo Jiang, Philippe Nain, and Don Towsley. On the convergence of the ttl approximation for an lru cache under independent stationary request processes. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 3(4), September 2018. 4

[164] S. Jiang and X. Zhang. ULC: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In *24th International Conference on Distributed Computing Systems, 2004. Proceedings.*, pages 168–177, March 2004. doi: 10.1109/ICDCS.2004.1281581. ISSN: 1063-6927. 8.6

[165] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *ACM SIGMETRICS Performance Evaluation Review*, volume 30 of *SIGMETRICS'02*, pages 31–42, June 2002. doi: 10.1145/511399.511340. 1.1, 1.3, 2.3.1, 6.4, 7.1, 7.4, 7.4, 8.1.2, 8.2.2, 8.4.2, 8.7, 9.1.2, 9.2.2

[166] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'05, page 35, USA, April 2005. USENIX Association. 4.5.1, 6.4, 7.1, 7.3, 9.1.2

[167] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality. In *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, volume 4 of *FAST'05*, pages 8–8, 2005. 7.1, 8.6

[168] Song Jiang, F. Petrini, Xiaoning Ding, and Xiaodong Zhang. A Locality-Aware Cooperative Cache Management Protocol to Improve Network File System Performance. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, ICDCS'06, pages 42–42, July 2006. doi: 10.1109/ICDCS.2006.9. ISSN: 1063-6927. 8.6

[169] Wenjie Jiang, Rui Zhang-Shen, Jennifer Rexford, and Mung Chiang. Cooperative content distribution and traffic engineering in an ISP network. In *Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, SIGMETRICS'09, pages 239–250, New York, NY, USA, June 2009. Association for Computing Machinery. ISBN 978-1-60558-511-6. doi: 10.1145/1555349.1555377. URL https://doi.org/10.1145/1555349.1555377. 8.6

[170] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. NetCache: Balancing key-value stores with fast in-network caching. In *Proceedings of the 26th symposium on operating systems principles*, SOSP'17, pages 121–136, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132764. 3.3.5, 7.1

[171] Alekh Jindal, Shi Qiao, Rathijit Sen, and Hiren Patel. Microlearner: A fine-grained learning optimizer for big data workloads at microsoft. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 2423–2434. IEEE, 2021. 6.4

[172] Theodore Johnson and Dennis Shasha. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB'94, pages 439–450, San Francisco, CA, USA, September 1994. Morgan Kaufmann Publishers Inc. ISBN 978-1-55860-153-6. 1.1, 1.3, 2.3.1, 6.4, 7.1, 7.3, 7.4, 8.1.2, 8.2.2, 8.4.2, 8.6, 9.1.2, 9.1.3, 9.2.2

[173] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. Dns performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pages 153–167, 2001. 3.3.4

[174] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'20, pages 369–385, 2020. ISBN 978-1-939133-19-9. URL https://www.usenix.org/conference/osdi20/presentation/kadekodi. 5.3.1

[175] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996. 6.1.1

[176] George Karakostas and D Serpanos. Practical LFU implementation for web caching. *Technical Report TR-622-00*, 2000. 4.2.6, 4.5.1, 6.4

[177] R. Karedla, J.S. Love, and B.G. Wherry. Caching strategies to improve disk system performance. *Computer*, 27(3):38–46, March 1994. ISSN 1558-0814. doi: 10.1109/2.268884. 7.1, 8.1.2, 8.4.2, 9.1.2

[178] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing*, pages 654–663, 1997. 2

[179] Hyojun Kim, Moonkyung Ryu, and Umakishore Ramachandran. What is a good

buffer cache replacement scheme for mobile flash storage? *ACM SIGMETRICS Performance Evaluation Review*, 40(1):235–246, 2012. 8.1, 8.6

[180] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *4th USENIX Symposium on Operating Systems Design and Implementation*, OSDI'00, USA, 2000. USENIX Association. 8.6

[181] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K. Sitaraman. RL-Cache: Learning-Based Cache Admission for Content Delivery. In *Proceedings of the 2019 Workshop on Network Meets AI & ML - NetAI'19*, NetAI'19, pages 57–63, Beijing, China, 2019. ACM Press. ISBN 978-1-4503-6872-8. doi: 10.1145/3341216.3342214. URL http://dl.acm.org/citation.cfm?doid=3341216.3342214. 6.4

[182] Ricardo Koller and Raju Rangaswami. I/o deduplication: Utilizing content similarity to improve i/o performance. *ACM Transactions on Storage (TOS)*, 6(3):1–26, 2010. 7.1, 8.1

[183] Jack Kosaian, K. V. Rashmi, and Shivaram Venkataraman. Parity models: erasure-coded resilience for prediction serving systems. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP'19, pages 30–46, New York, NY, USA, October 2019. Association for Computing Machinery. ISBN 978-1-4503-6873-5. doi: 10.1145/3341301.3359654. URL https://dl.acm.org/doi/10.1145/3341301.3359654. 6.4

[184] Jack Kosaian, Amar Phanishayee, Matthai Philipose, Debadeepta Dey, and Rashmi Vinayak. Boosting the Throughput and Accelerator Utilization of Specialized CNN Inference Beyond Increasing Batch Size. In *Proceedings of the 38th International Conference on Machine Learning*, pages 5731–5741. PMLR, July 2021. URL https://proceedings.mlr.press/v139/kosaian21a.html. ISSN: 2640-3498. 6.4

[185] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD'18, pages 489–504, New York, NY, USA, May 2018. Association for Computing Machinery. ISBN 978-1-4503-4703-7. doi: 10.1145/3183713.3196909. URL https://doi.org/10.1145/3183713.3196909. 6.4

[186] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35(11):190–201, November 2000. ISSN 0362-1340. doi: 10.1145/356989.357007. URL https://doi.org/10.1145/356989.357007. 5.7

[187] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Systor '17 traces (SNIA IOTTA trace 5102). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2016. URL http://iotta.snia.org/traces/block-io

/4964?only=5102. 8.1

[188] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR'17, pages 1–11, New York, NY, USA, May 2017. Association for Computing Machinery. ISBN 978-1-4503-5035-8. doi: 10.1145/3078468.3078479. URL https://dl.acm.org/doi/10.1145/3078468.3078479. 8.1

[189] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP'17, pages 137–152, Shanghai China, October 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132756. URL https://dl.acm.org/doi/10.1145/3132747.3132756. 3.8

[190] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and Analysis of a Container-Based Flash Cache for Compound Objects. *ACM Transactions on Storage*, 13(3):1–34, October 2017. ISSN 1553-3077, 1553-3093. doi: 10.1145/3094785. 5.7, 8.6

[191] Cong Li. DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics. In *Proceedings of the 11th ACM International Systems and Storage Conference*, SYSTOR'18, pages 59–64, New York, NY, USA, June 2018. Association for Computing Machinery. ISBN 978-1-4503-5849-1. doi: 10.1145/3211890.3211891. 6.4, 9.1.2

[192] Cong Li. CLOCK-pro+: improving CLOCK-pro cache replacement with utility-driven adaptation. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR'19, pages 1–7, New York, NY, USA, May 2019. Association for Computing Machinery. ISBN 978-1-4503-6749-3. doi: 10.1145/3319647.3325838. URL https://dl.acm.org/doi/10.1145/3319647.3325838. 8.6

[193] Conglong Li and Alan L. Cox. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys'15, pages 1–15, Bordeaux France, April 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741956. 2.3.1, 3.3.4, 4.1, 4.2.6, 6.4, 9.1.2

[194] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment*, 12(12):2118–2130, 2019. 6.4

[195] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'14, pages 1–15, Seattle WA USA, November 2014. ACM. ISBN 978-1-4503-3252-1. doi: 10.1145/2670979.2670985. URL https://dl.acm.org/doi/10.1145/2670979.2670985. 5.7

[196] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan,

Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'20, pages 591–605, New York, NY, USA, March 2020. Association for Computing Machinery. ISBN 978-1-4503-7102-5. doi: 10.1145/3373376.3378531. URL https://dl.acm.org/doi/10.1145/3373376.3378531. 8.1.1

[197] Jin Li and Cha Zhang. Distributed hosting of Web content with erasure coding and unequal weight assignment. In *2004 IEEE International Conference on Multimedia and Expo (ICME) (IEEE Cat. No.04TH8763)*, volume 3, pages 2087–2090 Vol.3, June 2004. doi: 10.1109/ICME.2004.1394677. URL https://ieeexplore.ieee.org/document/1394677. 5.7

[198] Jinhong Li, Qiuping Wang, Patrick PC Lee, and Chao Shi. An in-depth analysis of cloud block storage workloads in large-scale production. In *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pages 37–47. IEEE, 2020. 7.1, 8.1

[199] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA'15, pages 476–488, New York, NY, USA, June 2015. Association for Computing Machinery. ISBN 978-1-4503-3402-0. doi: 10.1145/2749469.2750416. URL https://dl.acm.org/doi/10.1145/2749469.2750416. 3.5.4, 4.1.3, 4.5.1

[200] 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)*, FAST'05, 2005. 8.6

[201] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. {C-Miner}: Mining Block Correlations in Storage Systems. 2004. 6.4, 8.6

[202] Yu Liang, Riwei Pan, Tianyu Ren, Yufei Cui, Rachata Ausavarungnirun, Xianzhang Chen, Changlong Li, Tei-Wei Kuo, and Chun Jason Xue. CacheSifter: Sifting Cache Files for Boosted Mobile Performance and Lifetime. In *20th USENIX Conference on File and Storage Technologies*, FAST'22, pages 445–459, 2022. 2.3.1, 7.1, 8.1

[203] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX symposium on networked systems design and implementation*, NSDI'14, pages 429–444, Seattle, WA, April 2014. USENIX Association. ISBN 978-1-931971-09-6. 1.1, 2.3.2, 3.1.4, 3.5.4, 3.8, 4.1.3, 4.1.4, 4.2.2, 4.2.3, 4.3.4, 4.5.2, 6.4, 8.3.2, 8.6, 9.1.2

[204] Kevin Lin. Improving distributed caching performance and efficiency at pinterest. https://medium.com/pinterest-engineering/improving-distributed-caching-performance-and-efficiency-at-pinterest-92484b5fe39b, 2024. Accessed: 2024-07-20. 1.1

[205] linux. The multi-generational lru. https://lwn.net/Articles/851184, 2024. Accessed: 2024-07-20. 8.1.1

[206] linux. Page frame reclamation. https://www.kernel.org/doc/gorman/html/understand/understand013.html, 2024. Accessed: 2024-07-20. 1.1, 7.1, 7.5

[207] lirs. Lirs implementation and discussion. https://github.com/facebook/CacheLib/discussions/99, 2024. Accessed: 2024-07-20. 8.4.1

[208] Evan 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*, pages 6237–6247. PMLR, November 2020. 6.1.1

[209] Xin Liu, Ashraf Aboulnaga, Kenneth Salem, and Xuhui Li. CLIC: CLient-Informed Caching for Storage Servers. In *FAST*, pages 297–310, 2009. 8.6

[210] Yudan Liu, Raja Nassar, Chokchai Leangsuksun, Nichamon Naksinehaboon, Mihaela Paun, and Stephen L. Scott. An optimal checkpoint/restart model for a large scale high performance computing system. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–9, April 2008. doi: 10.1109/IPDPS.2008.4536279. URL https://ieeexplore.ieee.org/abstract/document/4536279?casa_token=pZa2AQuIPesAAAAA:V_XcSBaNRxb8fZ7DkQLJeFR27_axcL6m9ededn1KbksapXMDQ7SiRgKpWdUTWLQM1G6W3tTObQ. ISSN: 1530-2075. 5.7

[211] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. DistCache: Provable load balancing for Large-Scale storage systems with distributed caching. In *17th USENIX conference on file and storage technologies*, FAST'19, pages 143–157, Boston, MA, February 2019. USENIX Association. ISBN 978-1-939133-09-0. 3.3.5, 3.6.3, 3.8, 7.1, 8.6

[212] lrb. Lrb implementation. https://github.com/sunnyszy/lrb, 2024. Accessed: 2024-07-20. 8.4.1

[213] lru-dict. lru-dict. https://github.com/amitdev/lru-dict, 2024. Accessed: 2023-04-27. 9.2.2, 9.2

[214] lru-rs. lru-rs. https://github.com/jeromefroe/lru-rs, 2024. Accessed: 2023-04-27. 9.2.2, 9.2

[215] Qiong Luo, Sailesh Krishnamurthy, C Mohan, Hamid Pirahesh, Honguk Woo, Bruce G Lindsay, and Jeffrey F Naughton. Middle-tier database caching for e-business. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 600–611, 2002. 3.1.4

[216] Dan Luu. A decade of major cache incidents at twitter. https://danluu.com/cache-incidents/, 2024. Accessed: 2024-07-28. 10.2.1

[217] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++

server workloads. In *25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020. 6.3.6

[218] Shashi Madappa. Ephemeral volatile caching in the cloud. `https://netflixtechblog.com/ephemeral-volatile-caching-in-the-cloud-8eba7b124589`, 2024. Accessed: 2024-07-20. 4.1.1, 4.4.2

[219] Bruce M. Maggs and Ramesh K. Sitaraman. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, July 2015. ISSN 0146-4833. doi: 10.1145/2805789.2805800. 2.3.1, 5.1, 5.2.3, 7.4, 7.5, 8.2, 8.2.1, 8.2.2

[220] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, Eurosys'12, page 183, Bern, Switzerland, 2012. ACM Press. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168855. URL `http://dl.acm.org/citation.cfm?doid=2168836.2168855`. 4.3.4, 4.5.1

[221] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711*, 2019. 6.4

[222] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers. *arXiv preprint arXiv:2004.03814*, 2020. 6.4

[223] Valentina Martina, Michele Garetto, and Emilio Leonardi. A unified approach to the performance analysis of caching systems. In *IEEE INFOCOM*, 2014. 5.3.3

[224] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, HPCA'22, pages 925–937, April 2022. doi: 10.1109/HPCA53966.2022.00072. ISSN: 2378-203X. 8.1, 8.6

[225] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*, SOSP'21, pages 243–262, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8709-5. doi: 10.1145/3477132.3483568. 2.1, 5.7, 8.1.1, 10.2.2

[226] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Theory and practice of caching billions of tiny objects on flash. In *ACM Transactions on Storage*, volume 18 of *TOS'22*, August 2022. doi: 10.1145/3542928. 7.1, 7.2, 8.3.2, 9.3.4

[227] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *2nd USENIX conference on file and storage technologies*, FAST'03, 2003. 1.1, 1.3, 2.3.1, 4.5.1, 6.2.5, 6.4, 7.1, 7.4, 7.4, 8.1.2, 8.2.2, 8.6, 9.1.2, 9.1.3, 9.2.2,

9.4.1, 9.7

[228] Memcached. Memcached - a distributed memory object caching system. http://memcached.org/, 2024. Accessed: 2024-07-20. 1.1, 3.1.1, 3.8, 4.2.3, 7.5, 8.1.2, 8.4.4, 9.1.1

[229] memcached. Lfu eviction policy. https://github.com/memcached/memcached/issues/543, 2024. Accessed: 2024-07-20. 4.1.1

[230] Memcached. Extstore. https://github.com/memcached/memcached/wiki/Extstore, 2024. Accessed: 2024-07-20. 8.1.1

[231] Memcached. Enhance slab reallocation for burst of evictions. https://github.com/memcached/memcached/pull/695, 2024. Accessed: 2024-07-20. 3.1.3

[232] Memcached. Experiencing slab ooms after one week of uptime. https://github.com/memcached/memcached/issues/689, 2024. Accessed: 2024-07-20. 3.1.3

[233] Memcached. Do not join lru and slab maintainer threads if they do not exist. https://github.com/memcached/memcached/pull/686, 2024. Accessed: 2024-07-20. 3.1.3

[234] Memcached. slab auto-mover anti-favours slab 2. https://github.com/memcached/memcached/issues/677, 2024. Accessed: 2024-07-20. 3.1.3

[235] memcached. Enhance slab reallocation for burst of eviction. https://github.com/memcached/memcached/pull/695, 2024. Accessed: 2024-07-20. 4.1.3

[236] memcached. Experiencing slab ooms after one week of uptime. https://github.com/memcached/memcached/issue/689, 2024. Accessed: 2024-07-20. 4.1.3

[237] memcached. slab auto-mover anti-favours slab 2. https://github.com/memcached/memcached/issue/677, 2024. Accessed: 2024-07-20. 4.1.3

[238] memcached. slabs: fix crash in page mover. https://github.com/memcached/memcached/pull/608, 2024. Accessed: 2024-07-20. 4.1.3

[239] memcached. bit vector + backoff timer + simpler implementation for faster slab reassignment. https://github.com/memcached/memcached/pull/542, 2024. Accessed: 2024-07-20. 4.1.3

[240] memcached. Faster slab reassignment. https://github.com/memcached/memcached/pull/524, 2024. Accessed: 2024-07-20. 4.1.3

[241] Memcached. Paper review: Memc3. https://memcached.org/blog/paper-review-memc3/, 2024. Accessed: 2024-07-20. 3.3.3

[242] memcached. Memcached benchmark. https://github.com/scylladb/seastar/wiki/Memcached-Benchmark, 2024. Accessed: 2024-07-20. 4.1.4

[243] Justin Meza, Tianyin Xu, Kaushik Veeraraghavan, and Onur Mutlu. A large scale study of data center network reliability. In *Proceedings of the Internet Measurement Conference 2018*, pages 393–407, 2018. 5.2.2

[244] MGLRU. Multi-gen lru. https://docs.kernel.org/admin-guide/mm/multigen_lru.html, 2024. Accessed: 2024-07-20. 7.1

[245] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent Hashing with Bounded Loads, July 2017. URL http://arxiv.org/abs/1608.01350. arXiv:1608.01350 [cs]. 5.7

[246] Vahab Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. Consistent hashing with bounded loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 587–604, 2018. URL http://epubs.siam.org/doi/abs/10.1137/1.9781611975031.39. 2

[247] mnemonist. mnemonist. https://github.com/Yomguithereal/mnemonist, 2024. Accessed: 2023-04-27. 9.2.2, 9.2

[248] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Caching in video CDNs: building strong lines of defense. In *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, EuroSys'14, pages 1–13, Amsterdam, The Netherlands, 2014. ACM Press. ISBN 978-1-4503-2704-6. doi: 10.1145/2592798.2592817. 8.1

[249] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's Warm BLOB Storage System. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 383–398, Broomfield, CO, October 2014. USENIX Association. ISBN 978-1-931971-16-4. URL https://www.usenix.org/conference/osdi14/technical-sessions/presentation/muralidhar. 5.3.1, 5.7

[250] MySQL. Mysql reference manual buffer pool. https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html, 2024. Accessed: 2024-07-20. 7.1, 7.5

[251] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. DeepCache: A Deep Learning Based Framework For Content Caching. In *Proceedings of the 2018 Workshop on Network Meets AI & ML*, NetAI'18, pages 48–53, New York, NY, USA, August 2018. Association for Computing Machinery. ISBN 978-1-4503-5911-5. doi: 10.1145/3229543.3229555. 6.1.1, 6.4

[252] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. MSR Cambridge traces (SNIA IOTTA trace set 388). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, March 2007. URL http://iotta.snia.org/traces/block-io?only=388. 6.3, 6.3.1, 7.1, 8.1

[253] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, San Jose, CA, February 2008. USENIX Association. URL https://www.usenix.org/conference/fast-08/write-loading-practical-power-management-enterprise-storage. 8.1

[254] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning Multi-Dimensional Indexes. In *Proceedings of the 2020 ACM SIGMOD International*

Conference on Management of Data, SIGMOD/PODS'20, pages 985–1000, Portland OR USA, June 2020. ACM. ISBN 978-1-4503-6735-6. doi: 10.1145/3318464.3380579. URL https://dl.acm.org/doi/10.1145/3318464.3380579. 6.4

[255] Netflix. Cache warming: Leveraging ebs for moving petabytes of data. https://netflixtechblog.medium.com/cache-warming-leveraging-ebs-for-moving-petabytes-of-data-adcf7a4a78c3, 2024. Accessed: 2024-07-20. 1.1

[256] Raymond Ng, Christos Faloutsos, and Timos Sellis. Flexible buffer allocation based on marginal gains. *ACM SIGMOD Record*, 20(2):387–396, April 1991. ISSN 0163-5808. doi: 10.1145/119995.115857. 8.6

[257] Yuanjiang Ni, Ji Jiang, Dejun Jiang, Xiaosong Ma, Jin Xiong, and Yuangang Wang. S-RAC: SSD Friendly Caching for Data Center Workloads. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR'16, pages 1–12, Haifa Israel, June 2016. ACM. ISBN 978-1-4503-4381-7. doi: 10.1145/2928275.2928284. 8.6

[258] Victor F. Nicola, Asit Dan, and Daniel M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *ACM SIGMETRICS Performance Evaluation Review*, volume 20 of *SIGMETRICS'92*, pages 35–46, June 1992. doi: 10.1145/149439.133084. 7.1, 7.3

[259] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, and others. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'13, pages 385–398, 2013. 1.2, 2.1, 3.1.1, 3.6.2, 3.6.5, 3.7, 3.8, 4.1.1, 4.1.4, 4.4.2, 5.1, 6.1.1, 8.1

[260] E. Nygren, Ramesh K. Sitaraman, and J. Sun. The Akamai Network: A platform for high-performance Internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010. 5.1, 5.2.2

[261] Erik Nygren, Ramesh K. Sitaraman, and Jennifer Sun. The Akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, August 2010. ISSN 0163-5980. doi: 10.1145/1842733.1842736. URL https://dl.acm.org/doi/10.1145/1842733.1842736. 8.1

[262] Yongseok Oh, Jongmoo Choi, Donghee Lee, and Sam H Noh. Caching less for better performance: balancing cache size and update cost of flash memory cache in hybrid storage systems. In *FAST*, volume 12, 2012. 5.7

[263] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record*, 22(2): 297–306, June 1993. ISSN 0163-5808. doi: 10.1145/170036.170081. 4.2.6, 4.5.1, 6.4, 8.2.2, 9.1.2, 9.1.3, 9.5.1

[264] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage

System. *ACM Transactions on Computer Systems*, 33(3):1–55, September 2015. ISSN 0734-2071, 1557-7333. doi: 10.1145/2806887. URL https://dl.acm.org/doi/10.1145/2806887. 1.2, 4.2.2, 4.5.2, 6.2.4

[265] John K. Ousterhout, Hervé Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the tenth ACM symposium on Operating systems principles*, SOSP '85, pages 15–24, New York, NY, USA, December 1985. Association for Computing Machinery. ISBN 978-0-89791-174-0. doi: 10.1145/323647.323631. URL https://doi.org/10.1145/323647.323631. 3.7

[266] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004. 4.1.2

[267] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC'19, pages 193–205, Santa Cruz CA USA, November 2019. ACM. ISBN 978-1-4503-6973-2. doi: 10.1145/3357223.3362729. 4.5.1, 8.1

[268] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *Proceedings of the 2020 USENIX Conference on Usenix Annual Technical Conference*, pages 813–827, 2020. 8.1.2

[269] Sejin Park and Chanik Park. FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *Proc. of the 33rd IEEE International Conference on Massive Storage Systems and Technology (MSST)*, 2017. 6.1.1, 9.1.2

[270] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, New York, NY, USA, June 1988. Association for Computing Machinery. ISBN 978-0-89791-268-6. doi: 10.1145/50202.50214. URL https://doi.org/10.1145/50202.50214. 5.7

[271] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP'95, pages 79–95, New York, NY, USA, December 1995. Association for Computing Machinery. ISBN 978-0-89791-715-5. doi: 10.1145/224056.224064. 8.6

[272] Laurent Perron and Vincent Furnon. OR-tools. https://developers.google.com/optimization/, 2023. 5.4

[273] Aidi Pi, Junxian Zhao, Shaoqi Wang, and Xiaobo Zhou. Memory at your service: Fast memory allocation for latency-critical services. In *Proceedings of the 22nd International Middleware Conference*, Middleware '21, pages 185–197, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9781450385343. doi: 10.1145/3464298.3493394. URL https://doi.org/10.1145/3464298.3493394. 2.3.2

[274] Timothy Pritchett and Mithuna Thottethodi. SieveStore: a highly-selective,

ensemble-level disk cache for cost-performance. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA'10, pages 163–174, New York, NY, USA, June 2010. Association for Computing Machinery. ISBN 978-1-4503-0053-7. doi: 10.1145/1815961.1815982. 8.6

[275] Paula Pufek, Hrvoje Grgić, and Branko Mihaljević. Analysis of garbage collection algorithms and memory management in java. In *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1677–1682. IEEE, 2019. 7.5

[276] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, EuroSys'23, pages 557–573, New York, NY, USA, May 2023. Association for Computing Machinery. ISBN 978-1-4503-9487-1. doi: 10.1145/3552326.3587446. 1.1, 7.1, 7.5, 8.6, 9.1.1, 9.1.2

[277] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD/PODS'17, pages 51–63, Chicago Illinois USA, May 2017. ACM. ISBN 978-1-4503-4197-4. doi: 10.1145/3035918.3056100. URL https://dl.acm.org/doi/10.1145/3035918.3056100. 5.7

[278] K. V Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, 2013. 5.2.2, 5.3.1

[279] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, HotStorage'13, 2013. URL https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/rashmi. 5.7

[280] KV Rashmi, Nihar B Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *SIGCOMM*, 2015. 5.2.2

[281] KV Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. EC-Cache:load-balanced,low-latency cluster caching with online erasure coding. In *12th USENIX symposium on operating systems design and implementation*, OSDI'16, pages 401–417, 2016. 3.3.5, 3.6.3, 5.5.5, 5.7, 7.1, 8.1, 8.6

[282] Redis. database caching strategy using redis. https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf, 2024. Accessed: 2024-07-20. 3.3.4, 4.1.1

[283] redis. Redis. http://redis.io/, 2024. Accessed: 2024-07-20. 3.8

[284] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. Google cluster-usage traces: format + schema. Technical report, Google Inc., Mountain View, CA, USA, November 2011. Revised 2014-11-17 for version 2.1. Posted at https://github.com/google/cluster-data. 3.8

[285] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *ACM Symposium on Cloud Computing (SoCC)*, San Jose, CA, USA, October 2012. URL http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-socc2012.pdf. 3.8

[286] Sean Rhea, Patrick Eaton, Dennis Geels, Hakim Weatherspoon, Ben Zhao, and John Kubiatowicz. Pond: The OceanStore Prototype. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, FAST'03, 2003. URL https://www.usenix.org/conference/fast-03/pond-oceanstore-prototype. 5.7

[287] John T. Robinson and Murthy V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on measurement and modeling of computer systems*, SIGMETRICS'90, pages 134–142, New York, NY, USA, 1990. Association for Computing Machinery. ISBN 0-89791-359-0. doi: 10.1145/98457.98523. 9.1.2

[288] RocksDB. Rocksdb. https://rocksdb.org/, 2024. Accessed: 2024-07-20. 4.5.2

[289] RocksDB. Lock-free clock cache. https://github.com/facebook/rocksdb/issues/10306, 2024. Accessed: 2024-12-06. 8.1.2

[290] Liana V. Rodriguez, Alexis Gonzalez, Pratik Poudel, Raju Rangaswami, and Jason Liu. Unifying the data center caching layer: Feasible? Profitable? In *Proceedings of the 13th ACM workshop on hot topics in storage and file systems*, HotStorage'21, pages 50–57, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 978-1-4503-8550-3. doi: 10.1145/3465332.3470884. 7.1, 8.1

[291] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies*, FAST'21, pages 341–354. USENIX Association, February 2021. ISBN 978-1-939133-20-5. 1.1, 2.3.1, 4.1, 6.1.1, 6.1.1, 6.2.3, 6.2.5, 6.3.1, 7.1, 7.4, 7.4, 7.6, 8.1.2, 8.2.2, 8.4.2, 8.6, 9.1.1, 9.1.2, 9.1.3, 9.2.2, 10.2.1

[292] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992. ISSN 0734-2071. doi: 10.1145/146941.146943. URL https://doi.org/10.1145/146941.146943. 1.2, 4.2.2, 4.5.2, 6.2.4

[293] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured Memory for DRAM-based Storage. In *12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, 2014. ISBN 978-1-931971-08-9. URL https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble. 2.3.2, 4.1, 4.1.2, 4.1.3, 4.2.2, 4.5.2, 6.2.4

[294] Amazon SageMaker. Xgboost algorithm. https://docs.aws.amazon.com/sagemaker/latest/dg/xgboost.html, 2024. Accessed: 2024-07-20. 6.2.5

[295] Ramendra K Sahoo, Mark S Squillante, Anand Sivasubramaniam, and Yanyong Zhang. Failure data analysis of a large-scale heterogeneous server environment. In *International Conference on Dependable Systems and Networks, 2004*, pages 772–781, 2004. 5.2.2

[296] Lorenzo Saino, Ioannis Psaras, and George Pavlou. Understanding sharded caching systems. In *IEEE INFOCOM*, pages 1–9, 2016. 5.3.3

[297] Prasenjit Sarkar and John Hartman. Efficient Cooperative Caching Using Hints. In *USENIX 2nd Symposium on OS Design and Implementation* (*OSDI 96*), OSDI96, 1996. 8.6

[298] Prasenjit Sarkar and John H. Hartman. Hint-based cooperative caching. *ACM Trans. Comput. Syst.*, 18(4):387–419, November 2000. ISSN 0734-2071. doi: 10.1145/362670.362675. 5.7

[299] Mohit Saxena, Michael M. Swift, and Yiying Zhang. FlashTier: a lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*, EuroSys'12, page 267, Bern, Switzerland, 2012. ACM Press. ISBN 978-1-4503-1223-3. doi: 10.1145/2168836.2168863. 5.7, 8.6

[300] Kyle Schomp, Onkar Bhardwaj, Eymen Kurdoglu, Mashooq Muhaimen, and Ramesh K. Sitaraman. Akamai DNS: Providing Authoritative Answers to the World's Queries. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, SIGCOMM'20, pages 465–478, New York, NY, USA, July 2020. Association for Computing Machinery. ISBN 978-1-4503-7955-7. doi: 10.1145/3387514.3405881. URL https://doi.org/10.1145/3387514.3405881. 5.6, 8.1

[301] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009. 5.2.2

[302] Dimitrios N Serpanos and Wayne H Wolf. Caching web objects using zipf's law. In *Multimedia Storage and Archiving Systems III*, volume 3527, pages 320–326. International Society for Optics and Photonics, 1998. 4.2.6

[303] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Gdpr anti-patterns. *Commun. ACM*, 64(2):59–65, January 2023. ISSN 0001-0782. doi: 10.1145/3378061. URL https://doi.org/10.1145/3378061. 4.1.1

[304] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. Optimizing flash-based key-value cache systems. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, HotStorage'16, 2016. 5.7

[305] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. DIDACache: An Integration of Device and Application for Flash-based Key-value Caching. *ACM Transactions on Storage*, 14(3):26:1–26:32, October 2018. ISSN 1553-3077. doi: 10.1145/3203410. 5.1, 5.7, 8.6

[306] Wanxin Shi, Qing Li, Chao Wang, Gengbiao Shen, Weichao Li, Yu Wu, and Yong Jiang. LEAP: learning-based smart edge with caching and prefetching for adaptive video streaming. In *Proceedings of the International Symposium on Quality of Service*, pages 1–10, 2019. 6.1.1

[307] Weisong Shi, Randy Wright, Eli Collins, and Vijay Karamcheti. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the Seventh International Workshop on Web Caching and Content Distribution (WCW'02)*, pages 1–16. Citeseer, 2002. 3.7

[308] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM'21, pages 93–105, Virtual Event USA, August 2021. ACM. ISBN 978-1-4503-8383-7. doi: 10.1145/3452296.3472934. 7.1

[309] Yannis Smaragdakis, Scott Kaplan, and Paul Wilson. EELRU: simple and effective adaptive page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 27 (1):122–133, May 1999. ISSN 0163-5999. doi: 10.1145/301464.301486. 7.1, 8.1.2, 8.2.2, 9.1.2, 9.1.3

[310] Alan Jay Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978. ISSN 0362-5915. doi: 10.1145/320263.320276. 7.1, 9.1.2

[311] Hyunsub Song, Shean Kim, J. Hyun Kim, Ethan JH Park, and Sam H. Noh. First Responder: Persistent Memory Simultaneously as High Performance Buffer Cache and Storage. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, ATC'21, pages 839–853, 2021. ISBN 978-1-939133-23-6. URL `https://www.usenix.org/conference/atc21/presentation/song`. 8.1

[312] Zhenyu Song, Daniel S Berger, Kai Li, Anees Shaikh, Wyatt Lloyd, Soudeh Ghorbani, Changhoon Kim, Aditya Akella, Arvind Krishnamurthy, Emmett Witchel, and others. Learning relaxed belady for content distribution network caching. In *17th USENIX symposium on networked systems design and implementation*, NSDI'20, pages 529–544, 2020. (document), 1.1, 1.3, 2.3.1, 4.1, 6.1.1, 6.1.1, 6.2.2, 6.2.5, 6.3, 6.3.1, 6.4, 7.1, 8.4.2, 8.5.1, 9.1.2, 9.6.1, 10.2.1

[313] Zhenyu Song, Kevin Chen, Nikhil Sarda, Deniz Altinbuken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi. HALP:

Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network. In *20th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'23, pages 1149–1163, 2023. ISBN 978-1-939133-33-5. 9.1.2, 9.5.1

[314] Gokul Soundararajan, Madalin Mihailescu, and Cristiana Amza. Context-aware prefetching at the storage server. In *USENIX 2008 Annual Technical Conference*, ATC'08, pages 377–390, USA, June 2008. USENIX Association. 6.4, 8.6

[315] Kunwadee Sripanidkulchai, Bruce Maggs, and Hui Zhang. An analysis of live streaming workloads on the internet. In *Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, IMC '04, pages 41–54, New York, NY, USA, October 2004. Association for Computing Machinery. ISBN 978-1-58113-821-4. doi: 10.1145/1028788.1028795. URL https://doi.org/10.1145/1028788.1028795. 9.1.1

[316] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. *arXiv preprint arXiv:2301.00072*, 2022. 6.4

[317] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. Footprint Descriptors: Theory and Practice of Cache Provisioning in a Global CDN. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT'17, pages 55–67, Incheon Republic of Korea, November 2017. ACM. ISBN 978-1-4503-5422-6. doi: 10.1145/3143361.3143368. 5.2.1, 7.1, 8.1, 9.1.1

[318] Aditya Sundarrajan, Mangesh Kasbekar, Ramesh K. Sitaraman, and Samta Shukla. Midgress-aware traffic provisioning for content delivery. In *2020 USENIX Annual Technical Conference*, ATC'20, pages 543–557, 2020. ISBN 978-1-939133-14-4. 5.1

[319] Richard S Sutton. Introduction: The challenge of reinforcement learning. In *Reinforcement Learning*, pages 1–3. Springer, 1992. 6.1.1

[320] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 373–386, 2015. 2.1, 2.3.2, 3.1.4, 4.2.6, 5.1, 5.2.4, 5.5.7, 5.7, 7.1, 8.1.1, 8.6, 9.1.2, 10.2.2

[321] Matthew Tejo. Improving key expiration in redis. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/improving-key-expiration-in-redis.html, 2024. Accessed: 2024-07-20. 4.1.1, 4.4.2

[322] trau. Cache implementation in python. https://github.com/trauzti/cache, 2024. Accessed: 2024-07-20. 8.4.1

[323] Rollins Turner and Henry Levy. Segmented FIFO page replacement. *ACM SIGMETRICS Performance Evaluation Review*, 10(3):48–51, September 1981. ISSN 0163-5999. doi: 10.1145/1010629.805473. 8.6

[324] Twitter. twitter twemcache. https://github.com/twitter/twemcache, 2024. Accessed: 2024-07-20. 3.8, 4.2.3

[325] Twitter. Decomposing twitter: Adventures in service-oriented architecture. https://www.infoq.com/presentations/twitter-soa/, 2024. Accessed: 2024-07-25. 3.1.1

[326] Cristian Ungureanu, Biplob Debnath, Stephen Rago, and Akshat Aranya. TBF: A memory-efficient replacement policy for flash-based caches. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, ICDE'13, pages 1117–1128, April 2013. doi: 10.1109/ICDE.2013.6544902. ISSN: 1063-6382. 8.6

[327] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*, pages 1009–1024, 2017. 6.4

[328] Varnish. Varnish cache. https://varnish-cache.org/, 2024. Accessed: 2024-07-20. 3.5.1, 5.5.7

[329] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX workshop on hot topics in storage and file systems*, hotStorage'18, Boston, MA, July 2018. USENIX Association. 1.1, 1.3, 2.3.1, 6.1.1, 6.1.1, 7.1, 7.4, 7.4, 8.1.2, 8.2.2, 8.4.2, 9.1.2, 9.2.2, 9.5.1

[330] Vimeo. Galaxy cache. hhttps://github.com/vimeo/galaxycache, 2024. Accessed: 2024-07-20. 8.1.2

[331] Geoffrey M. Voelker, Eric J. Anderson, Tracy Kimbrel, Michael J. Feeley, Jeffrey S. Chase, Anna R. Karlin, and Henry M. Levy. Implementing cooperative prefetching and caching in a globally-managed memory system. In *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS'98/PERFORMANCE '98, pages 33–43, New York, NY, USA, June 1998. Association for Computing Machinery. ISBN 978-0-89791-982-1. doi: 10.1145/277851.277869. 8.6

[332] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *2017 USENIX annual technical conference*, ATC'17, pages 487–498, Santa Clara, CA, July 2017. USENIX Association. ISBN 978-1-931971-38-6. 5.5.1, 8.5.2, 9.1.2

[333] 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. ISBN 978-1-931971-20-1. 5.5.1, 6.3, 6.3.1, 7.1, 8.1, 8.5.2

[334] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. Learning caching policies with subsampling. In *NeurIPS Machine Learning for Systems Workshop*, 2019. 6.4

[335] Hua Wang, Xinbo Yi, Ping Huang, Bin Cheng, and Ke Zhou. Efficient SSD Caching by Avoiding Unnecessary Writes using Machine Learning. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP'18, pages 1–10, Eugene OR

USA, August 2018. ACM. ISBN 978-1-4503-6510-9. doi: 10.1145/3225058.3225126. 8.6, 9.1.2

[336] Hua Wang, Jiawei Zhang, Ping Huang, Xinbo Yi, Bin Cheng, and Ke Zhou. Cache What You Need to Cache: Reducing Write Traffic in Cloud Cache via "One-Time-Access-Exclusion" Policy. *ACM Transactions on Storage*, 16(3):1–24, August 2020. ISSN 1553-3077, 1553-3093. doi: 10.1145/3397766. 8.6

[337] Jaylen Wang, Daniel S. Berger, Fiodar Kazhamiaka, Celine Irvene, Chaojie Zhang, Esha Choukse, Kali Frost, Rodrigo Fonseca, Brijesh Warrier, Chetan Bansal, Jonathan Stern, Ricardo Bianchini, and Akshitha Sriraman. Designing Cloud Servers for Lower Carbon. In *ISCA*, June 2024. URL https://www.microsoft.com/en-us/research/publication/designing-cloud-servers-for-lower-carbon/. 10.2.2

[338] Joseph Wang, Anne Holler, Mingshi Wang, and Michael Mui. Productionizing distributed xgboost to train deep tree models with large data sets at uber. https://eng.uber.com/productionizing-distributed-xgboost, 2024. Accessed: 2024-07-20. 6.2.5

[339] Limin Wang, Kyoung Soo Park, Ruoming Pang, Vivek Pai, and Larry Peterson. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC'04, page 14, USA, June 2004. USENIX Association. 5.7

[340] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick PC Lee. Austere flash caching with deduplication and compression. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, ATC'20, pages 713–726, 2020. 7.1

[341] Qiuping Wang, Jinhong Li, Tao Ouyang, Chao Shi, and Lilong Huang. Separating data via block invalidation time inference for write amplification reduction in {Log-Structured} storage. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 429–444, 2022. 7.1, 8.1

[342] Hakim Weatherspoon and John D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In Gerhard Goos, Juris Hartmanis, Jan Van Leeuwen, Peter Druschel, Frans Kaashoek, and Antony Rowstron, editors, *International Workshop on Peer-to-Peer Systems*, volume 2429, pages 328–337, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg. URL http://link.springer.com/10.1007/3-540-45748-8_31. 5.7

[343] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 307–320, USA, November 2006. USENIX Association. ISBN 978-1-931971-47-8. URL https://dl.acm.org/doi/abs/10.5555/1298455.1298485. 5.7

[344] Patrick Wendell and Michael J. Freedman. Going viral: flash crowds in an open CDN. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement*

conference - IMC '11, IMC'11, page 549, Berlin, Germany, 2011. ACM Press. ISBN 978-1-4503-1013-0. doi: 10.1145/2068816.2068867. URL http://dl.acm.org/citation.cfm?doid=2068816.2068867. 3.7

[345] Alex Wiggins and Jimmy Langston. Enhancing the scalability of memcached. *Intel document*, 2012. 7.5

[346] wiki. Approximate counting algorithm. https://en.wikipedia.org/wiki/Approximate_counting_algorithm, 2023. Accessed: 2023-02-06. 4.2.6

[347] Wikimedia. caching overview - wikitech. https://wikitech.wikimedia.org/wiki/Caching_overview, 2024. Accessed: 2024-07-20. 3.1.1, 5.6

[348] WikiMedia. Better handling for one-hit-wonder objects. https://phabricator.wikimedia.org/T144187, 2024. Accessed: 2024-07-20. 2.3.1, 7.4, 8.2, 8.2.2

[349] wikimedia. Analytics/data lake/traffic/caching. https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching, 2024. Accessed: 2024-07-20. 3.8, 7.1, 8.4.4, 9.3.1, 9.1

[350] John Wilkes. More Google cluster data. Google research blog, November 2011. Posted at http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html. 3.8

[351] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference*, ATC'02, pages 161–175, 2002. 8.6, 9.4.3

[352] Guanying Wu and Xubin He. Reducing ssd read latency via nand flash program and erase suspension. In *FAST*, volume 12, pages 10–10, 2012. 5.2.4

[353] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association. URL https://www.usenix.org/conference/hotstorage19/presentation/wu-kan. 5.2.4

[354] 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. ISBN 978-1-939133-20-5. 7.1

[355] Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen, Kwanghyun Park, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. NyxCache: Flexible and Efficient Multi-tenant Persistent Memory Caching. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, FAST'22, pages 1–16, 2022. ISBN 978-1-939133-26-7. URL https://www.usenix.org/conference/fast22/presentation/wu. 8.1

[356] Nan Wu and Pengcheng Li. Phoebe: Reuse-Aware Online Caching with Reinforcement Learning for Emerging Storage Models, November 2020. 6.1.1, 9.1.2

[357] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zExpander: a key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, Eurosys'16, pages 1–15, London United Kingdom, April 2016. ACM. ISBN 978-1-4503-4240-7. doi: 10.1145/2901318.2901332. 4.5.1, 8.6

[358] xgboost. Xgboost. `https://github.com/dmlc/xgboost`, 2024. Accessed: 2024-07-20. 6.3.1

[359] Fangliang Xu, Yijie Wang, and Xingkong Ma. Online Encoding for Erasure-Coded Distributed Storage Systems. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 338–342, June 2017. doi: 10.1109/ICDCSW.2017.45. URL `https://ieeexplore.ieee.org/document/7979843`. ISSN: 2332-5666. 5.7

[360] Gala Yadgar, Michael Factor, and Assaf Schuster. Karma: Know-it-All Replacement for a Multilevel Cache. In *5th USENIX Conference on File and Storage Technologies (FAST 07)*, FAST'07, 2007. 8.6

[361] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. MC2: Multiple Clients on a Multilevel Cache. In *2008 The 28th International Conference on Distributed Computing Systems*, pages 722–730, June 2008. doi: 10.1109/ICDCS.2008.29. ISSN: 1063-6927. 8.6

[362] Gala Yadgar, Michael Factor, Kai Li, and Assaf Schuster. Management of Multilevel, Multiclient Cache Hierarchies with Application Hints. *ACM Transactions on Computer Systems*, 29(2):5:1–5:51, May 2011. ISSN 0734-2071. doi: 10.1145/1963559.1963561. 8.6

[363] Gala Yadgar, Michael Factor, and Assaf Schuster. Cooperative caching with return on investment. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST, pages 1–13, May 2013. doi: 10.1109/MSST.2013.6558446. ISSN: 2160-1968. 8.6

[364] Gang Yan and Jian Li. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proceedings of the 28th ACM International Conference on Multimedia*, MM'20, pages 1009–1017, Seattle WA USA, October 2020. ACM. ISBN 978-1-4503-7988-5. doi: 10.1145/3394171.3413524. 6.1.1, 6.2.5, 8.1, 9.1.2

[365] Gang Yan and Jian Li. Towards Latency Awareness for Content Delivery Network Caching. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 789–804, 2022. 8.1, 9.1.1

[366] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in nand ssds. *ACM Trans. Storage*, 13 (3), October 2017. 5.5.4

[367] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM*

Transactions on Storage, 13(3):1–26, October 2017. ISSN 1553-3077, 1553-3093. doi: 10.1145/3121133. URL https://dl.acm.org/doi/10.1145/3121133. 5.2.4, 8.1.1

[368] Juncheng Yang. libcachesim. http://github.com/1a1a11a/libcachesim, 2024. Accessed: 2024-07-20. 3.5.2, 6.3.1, 8.4.1, 9.2.2, 9.3.1

[369] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC'17, pages 66–79, New York, NY, USA, September 2017. Association for Computing Machinery. ISBN 978-1-4503-5028-0. doi: 10.1145/3127479.3131210. 3.1.4, 4.1, 4.5.1, 6.1.1, 6.1.1, 6.4, 8.6

[370] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX symposium on operating systems design and implementation*, OSDI'20, pages 191–208. USENIX Association, November 2020. ISBN 978-1-939133-19-9. 1.1, 1.2, 4.1.1, 4.1.2, 4.1.3, 4.2.4, 4.3.2, 4.4.2, 5.1, 6.1.1, 6.2.2, 7.1, 7.2, 7.1, 7.4, 8.2.1, 8.1, 9.1.1, 9.3.1, 9.1, 9.3.2, 9.3.4, 9.4.3, 9.6.3

[371] Juncheng Yang, Yao Yue, and K. V. Rashmi. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Transactions on Storage*, 17 (3):17:1–17:35, August 2021. ISSN 1553-3077. doi: 10.1145/3468521. 1.2

[372] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'21, pages 503–518. USENIX Association, April 2021. ISBN 978-1-939133-21-2. 1.2, 6.2.2, 6.2.3, 6.2.6, 6.3.1, 6.3.3, 6.4, 7.1, 7.2, 8.1.2, 8.3.2, 8.4.1, 8.4.2, 8.4.3, 8.6, 9.1.1, 9.1.2, 9.3.2, 9.3.3, 9.6.3

[373] Juncheng Yang, Anirudh Sabnis, Daniel S. Berger, K. V. Rashmi, and Ramesh K. Sitaraman. C2DN: How to harness erasure codes at the edge for efficient content delivery. In *19th USENIX symposium on networked systems design and implementation*, NSDI'22, pages 1159–1177, Renton, WA, April 2022. USENIX Association. ISBN 978-1-939133-27-4. 1.2, 5.3.3, 7.1, 8.1, 9.1.1

[374] Juncheng Yang, Ziming Mao, Yao Yue, 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)*, FAST'23, pages 115–134, 2023. ISBN 978-1-939133-32-8. 1.2, 9.1.2

[375] Juncheng Yang, Ziyue Qiu, Yazhuo Zhang, Yao Yue, and K. V. Rashmi. FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS'23, pages 70–79, New York, NY, USA, June 2023. Association for Computing Machinery. ISBN 9798400701955. doi: 10.1145/3593856.3595887. 8.3.1, 8.6, 8.7, 9.1.3, 9.4.1, 9.6.1

[376] 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, pages 130–149, New York, NY, USA,

October 2023. Association for Computing Machinery. ISBN 9798400702297. doi: 10.1145/3600006.3613147. 9.6.1

[377] Lei Yang, Hong Wu, Tieying Zhang, Xuntao Cheng, Feifei Li, Lei Zou, Yujie Wang, Rongyao Chen, Jianying Wang, and Gui Huang. Leaper: a learned prefetcher for cache invalidation in LSM-tree based storage engines. *Proc. VLDB Endow.*, 13(12): 1976–1989, July 2020. ISSN 2150-8097. doi: 10.14778/3407790.3407803. 6.4

[378] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission Optimization for Google Datacenter Flash Caches. In *2022 USENIX Annual Technical Conference*, ATC'22, pages 1021–1036, Carlsbad, CA, July 2022. USENIX Association. ISBN 978-1-939133-29-15. 7.1, 7.2, 8.1.1, 8.4.4

[379] ycombinator. Expiration in redis 6. https://news.ycombinator.com/item?id=19664483, 2023. Accessed: 2023-02-06. 4.1.1

[380] Matt M. T. Yiu, Helen H. W. Chan, and Patrick P. C. Lee. Erasure coding for small objects in in-memory KV storage. In *Proceedings of the 10th ACM International Systems and Storage Conference*, SYSTOR '17, pages 1–12, New York, NY, USA, May 2017. Association for Computing Machinery. ISBN 978-1-4503-5035-8. doi: 10.1145/3078468.3078470. URL https://doi.org/10.1145/3078468.3078470. 5.7

[381] Yao Yue. Eviction strategies. https://github.com/twitter/twemcache/wiki/Eviction-Strategies, 2024. Accessed: 2024-07-20. 4.3.2

[382] Heng Zhang, Mingkai Dong, and Haibo Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, FAST'16, pages 167–180, 2016. ISBN 978-1-931971-28-7. URL https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng. 5.7

[383] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. Optimal Data Placement for Heterogeneous Cache, Memory, and Storage Systems. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, volume 4 of *SIGMETRICS'20*, pages 1–27, May 2020. doi: 10.1145/3379472. 7.1, 8.1

[384] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. When is the Cache Warm? Manufacturing a Rule of Thumb. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, HotCloud20, 2020. 8.1

[385] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K. V. Rashmi. {SIEVE} is Simpler than {LRU}: an Efficient {Turn-Key} Eviction Algorithm for Web Caches. pages 1229–1246, 2024. ISBN 978-1-939133-39-7. 1.2

[386] 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, pages 59–72, USA, February 2013. USENIX Association. 9.1.1

[387] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. Tencent block storage traces (SNIA IOTTA trace set 27917). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, October 2018. URL `http://iotta.snia.org/traces/parallel?only=27917`. 7.1

[388] Yu Zhang, Ke Zhou, Ping Huang, Hua Wang, Jianying Hu, Yangtao Wang, Yongguang Ji, and Bin Cheng. A Machine Learning Based Write Policy for SSD Cache in Cloud Block Storage. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, DATE'20, pages 1279–1282, Grenoble, France, March 2020. IEEE. ISBN 978-3-9819263-4-7. doi: 10.23919/DATE48585.2020.9116539. 8.6

[389] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. OSCA: An Online-Model based cache allocation scheme in cloud block storage systems. In *2023 U2ENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798. USENIX Association, July 2023. ISBN 978-1-939133-14-4. URL `https://www.usenix.org/conference/atc20/presentation/zhang-yu`. 7.1, 8.1

[390] Yuchao Zhang, Pengmiao Li, Zhili Zhang, Bo Bai, Gong Zhang, Wendong Wang, Bo Lian, and Ke Xu. AutoSight: Distributed Edge Caching in Short Video Network. *IEEE Network*, 34(3):194–199, May 2020. ISSN 0890-8044, 1558-156X. doi: 10.1109/MNET.001.1900345. 6.1.1

[391] Zhe Zhang, Amey Deshpande, Xiaosong Ma, Eno Thereska, and Dushyanth Narayanan. Does erasure coding have a role to play in my data center? May 2010. URL `https://www.microsoft.com/en-us/research/publication/does-erasure-coding-have-a-role-to-play-in-my-data-center/`. 5.7

[392] Zehua Zhao, Yan Ma, and Qun Cong. GDSF-Based Low Access Latency Web Proxy Caching Replacement Algorithm. In *Proceedings of the 2018 2nd International Conference on Computer Science and Artificial Intelligence - CSAI '18*, CSAI'18, pages 232–236, Shenzhen, China, 2018. ACM Press. ISBN 978-1-4503-6606-9. doi: 10.1145/3297156.3297237. 6.4

[393] Chen Zhong, Xingsheng Zhao, and Song Jiang. LIRS2: an improved LIRS replacement algorithm. In *Proceedings of the 14th ACM International Conference on Systems and Storage*, SYSTOR'21, pages 1–12, Haifa Israel, June 2021. ACM. ISBN 978-1-4503-8398-1. doi: 10.1145/3456727.3463772. 6.4, 7.1, 8.6, 9.1.2

[394] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenji Liu, and Tianming Yang. Tencent photo cache traces (SNIA IOTTA trace set 27476). In Geoff Kuenning, editor, *SNIA IOTTA Trace Repository*. Storage Networking Industry Association, February 2016. URL `http://iotta.snia.org/traces/parallel?only=27476`. 7.1, 9.3.1, 9.1

[395] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale:

A tencent case study. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, pages 284–294, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357838. doi: 10.1145/3205289.3205299. URL `https://doi.org/10.1145/3205289.3205299`. 8.1, 8.4.4, 9.3.1

[396] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, June 2004. ISSN 1558-2183. doi: 10.1109/TPDS.2004.13. 9.1.2

[397] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATC'01, pages 91–104, USA, 2001. USENIX Association. ISBN 1-880446-09-X. 1.1, 2.3.1, 4.5.1, 7.1, 7.4, 8.6