2024

DYNAMIC MEMORY MANAGEMENT FOR KEY-VALUE STORE

Yuchen Wang
Michigan Technological University, yuchenwa@mtu.edu

DYNAMIC MEMORY MANAGEMENT FOR KEY-VALUE STORE

By

Yuchen Wang

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY

2024

This dissertation has been approved in partial fulfillment of the requirements for the Degree of DOCTOR OF PHILOSOPHY in Computer Science.

Department of Computer Science

Dissertation Advisor:	*Dr. Zhenlin Wang*
Committee Member:	*Dr. Soner Onder*
Committee Member:	*Dr. Jianhui Yue*
Committee Member:	*Dr. Xiaoyong Yuan*
Department Chair:	*Dr. Zhenlin Wang*

Dedication

To my parents, wife, and children

for their love, support, and encouragement!

Contents

List of Figures

List of Tables

Preface

Chapter 3 contains material that is published in Proceedings of the 2020 International Symposium on Memory Systems (MemSys'20), September 2020 [1].

Chapter 4 contains material that is published in IEEE Transactions on Cloud Computing, October 2023 [2].

Acknowledgments

I am profoundly grateful to my advisor, Dr. Zhenlin Wang, whose unwavering guidance and support have been pivotal throughout my doctoral journey. He not only imparted academic wisdom but also infused my Ph.D. research with boundless enthusiasm and motivation. His deep insights and strategic advice were instrumental in navigating the complexities of my studies and in the successful completion of my dissertation. I am particularly thankful for his generosity in sharing valuable ideas, time, and funding, which significantly enhanced my research experience. Dr. Wang has been more than an advisor; he has been a mentor whose influence has shaped my academic and life perspective profoundly. I am truly fortunate to have had him guide my Ph.D. journey.

I would like to express my gratitude to my committee members, Dr. Soner Onder, Dr. Jianhui Yue, and Dr. Xiaoyong Yuan, for their invaluable feedback on my research. Their expertise and dedication significantly contributed to refining my work, and their thoughtful comments have been instrumental in my academic growth. I deeply appreciate the time and knowledge they have shared with me.

I am grateful to Junyao Yang for his consistent enthusiasm and patience throughout our collaborative research endeavors. His willingness to engage in deep, thoughtful

discussions about various research challenges has been invaluable. Junyao's creative insights and extensive expertise enriched our projects. I deeply appreciate his steadfast support and companionship throughout my academic journey.

I also would like to extend my deepest appreciation to Dr. Ruihong Zhang, Shuhan Xu, Dr. Yu Cai, Dr. Jean Mayo, Dr. Niusen Chen, and Dr. Daniel Byrne for their invaluable support in both my life and work.

Abstract

To minimize the latency of accessing back-end servers, modern web services often use in-memory key-value (k-v) stores at the front end to cache frequently accessed objects. Due to the limited memory capacity, these stores must be configured with a fixed amount of memory. Consequently, cache replacement is required when the footprint of the accessed objects exceeds the cache size.

This thesis presents a comprehensive exploration of advanced dynamic memory management techniques for k-v stores. The first study conducts a detailed analysis of K-LRU, a random sampling-based replacement policy, proposing a dynamic K configuration scheme to exploit the potential miss ratio gap among various Ks. Experimental results demonstrate a throughput improvement of up to 32.5% over the default static K setting.

Building on this, the second study extends the exploration of K-LRU to a multi-tenant k-v store environment, introducing a locality- and latency-aware memory partitioning scheme. This approach significantly enhances performance, achieving up to a 50.2% reduction in average access latency and a 262.8% increase in throughput compared to standard Redis. When compared to a state-of-the-art memory allocation design, the proposed scheme shows improvements of up to 24.8% in average access latency

and 61.8% in throughput.

Finally, inspired by emerging Compute Express Link (CXL) memory-sharing techniques, the third study pushes k-v store memory management into a multi-tier memory environment. This involves designing a software-defined tiered memory management architecture on top of a CXL memory-sharing switch. By dynamically identifying hot application data, efficiently migrating items among memory tiers based on popularity, and implementing multi-tenant memory partitioning, the proposed sdTMM system effectively integrates fast local DRAM with slower but larger CXL-shared memory. Evaluations across various workloads show that, even with 80% of the fast memory replaced by CXL-shared slow memory, sdTMM maintains an average performance impact of 13%, with the best-case impact as low as 2.2% compared to an all-fast memory over-provisioned system.

This research collectively advances the techniques of dynamic memory management, demonstrating promising performance improvements in k-v stores.

Chapter 1

Introduction

To reduce the latency of accessing back-end servers, today's web services widely use in-memory key-value (k-v) stores at the front end to cache frequently accessed objects. For large-scale web services, key-value stores like Redis and Memcached are crucial for ensuring low-latency service when handling massive workloads. Unlike a dedicated cache that serves a single application, a multi-tenant cache allows multiple applications to share a single cache instance, with the available memory partitioned to meet each tenant's caching requirements. Due to the limited memory size, an in-memory key-value store needs to be configured with a fixed amount of memory, making efficient memory utilization essential for optimal system performance.

1.1 Motivation and Research Problem

There are two directions for improving cache performance, one is to optimize the replacement policy under the cache size limit, and the other is to increase the available memory with the help of emerging low-cost, high-density memory-sharing techniques.

Regarding the first direction, research on hotspot issues indicates that accesses in real-world commercial k-v stores follow a power-law distribution, with popular keys dominating the accesses [3]. LAMA's evaluation of the Facebook ETC workload also shows high data locality [4]. Thus, LRU (Least Recently Used) is a good choice as it effectively exploits this locality. However, implementing exact LRU can be costly. In software caches, prioritizing items according to their last access time typically relies on linked structures to maintain their order [5], and evictions require list operations, including pointer updates. These processes introduce both space and computation overhead. Additionally, each access necessitates locking the LRU list to update the corresponding LRU priority, further degrading performance [6].

To avoid the expense of ordered data structures and enhance performance, many existing schemes have adopted random sampling. During eviction, according to a specified sampling configuration, a small number of K keys are randomly sampled from all keys in the memory, and the one with the lowest priority is evicted. We

term this eviction policy as K-LRU. Ideally, the item evicted from this small random sample closely approximates the lowest priority item in the entire cache [5]. This random sampling-based policy is lightweight and shows its flexibility. We observe that there can exist a significant miss ratio gap between exact LRU and random sampling-based LRU under different sampling size Ks. In this thesis, we first explore the configurable random sampling size K to improve single-tenant in-memory cache performance. Subsequently, we extend our work to a multi-tenant system by developing a new memory arbitration scheme that guides the memory allocation among the tenants and dynamically customizes K to better exploit the locality of each individual tenant.

As research continues to enhance the performance of in-memory k-v stores within the constraints of local DRAM capacity, web services and applications are expanding their memory demands and workload volumes in the era of AI training, cloud computing, and big data. For these k-v stores operating in the cloud, it's crucial to augment memory capacity and minimize access latency. Emerging technologies such as non-volatile memory (NVM) and memory-sharing offer increased capacity and lower costs per gigabyte, but they also present higher access latencies. Therefore, a multi-tiered memory system that combines the rapid access of local DRAM with the larger capacity of slower memory sources (like shared memory from other devices, NVM, and flash) could potentially optimize both the performance and cost-efficiency of k-v store deployments.

Typically, tiered memory systems can be structured in two main ways. The first method involves arranging different types of memory vertically, where the fast memory tier serves as a cache for the slow tiers. In this configuration, the k-v store system can leverage established cache management policies, with data movement managed by either dedicated hardware logic or the operating system. The second approach places the fast and slow memory tiers on the same level within the memory hierarchy. Here, a middleware software library or specialized memory management software is used to allocate user data across the memory tiers and handle data migrations. This horizontal arrangement presents more complexities and possibilities for tiered memory management due to potential limitations in OS-level memory migration tools. However, this method allows for a more nuanced integration of application-level workload patterns into the strategies for data placement and movement between the tiers.

Designing an effective tiered memory management system in a multi-tenant environment involves several challenges. This thesis presents a scheme that addresses key challenges, including accurately identifying performance-critical data to allocate to the fast memory tier, effectively partitioning memory across different tenants, optimizing the use of both fast and slow memory tiers, and minimizing the overhead associated with data migration between these tiers. These factors are crucial to maintaining high performance and efficiency in environments where resources are shared among multiple users.

1.2 Research Contributions

Motivated by the aforementioned directions for performance improvement in k-v stores, we made the following main contributions in this research.

1. We conduct a detailed analysis of the K-LRU behavior in Redis, identifying the potential for different miss ratios based on varying K values. We propose a dynamic configuration scheme, DLRU, that adopts a low-overhead miniature cache simulator and a cost model to predict miss ratios and optimize performance trade-offs. Our experiments show that DLRU can always match the performance of the best K setting, and improve the overall Redis throughput over the default K configuration by up to 32.5%.

2. Extending the exploration of K-LRU, we propose kRedis, a multi-tenant memory partition system, which is guided by merged K-LRU MRCs and is aware of tenant miss latency. This new memory arbitration scheme dynamically configures the random sampling size K for each individual tenant to adapt to access pattern changes, exploring the possible miss ratio gap between various K options. We adopt a new multi-dictionary design to increase the efficiency of random sampling for the individual tenants. The performance evaluation shows that kRedis attains up to a 50.2% lower average access latency, and up to a

262.8% higher hit throughput than Redis. When compared to pRedis, a state-of-the-art design of memory allocation, kRedis yields up to 24.8% and 61.8% improvements in access latency and throughput, respectively.

3. Inspired by the emerging Compute Express Link (CXL) memory-sharing techniques, we present sdTMM, a software-defined tiered memory management architecture. This system integrates fast local DRAM with slower, yet higher-capacity CXL-shared memory to create an efficient multi-tier memory pool. By dynamically identifying and placing hot data, efficiently migrating items among memory tiers based on their popularity, and implementing locality-aware multi-tenant memory partitioning, sdTMM optimizes memory utilization and maintains high performance. Our evaluations show that sdTMM, even with 80% of the fast memory replaced by CXL-shared slow memory, has an average performance impact of 13%, and the best-case performance impact of only 2.2% compared to an all-fast memory over-provisioned system.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows.

Chapter 2 reviews the background knowledge and discusses related research work in key-value stores and tiered memory management.

Chapter 3 proposes DLRU which explores the configuration of K in K-LRU, to improve the overall k-v system performance. We describe our system design, implementation, and experimental results.

Chapter 4 presents kRedis, a reference locality- and latency-aware memory partitioning scheme, to improve the performance of the in-memory multi-tenant k-v store that utilizes random sampling-based replacement algorithm. We show that our design significantly boosts system performance.

Chapter 5 proposes sdTMM, a software-defined tiered memory management scheme for in-memory k-v stores in the multi-tenant environment. We first present the system architecture and describe the main feature designs in detail. Then we discuss the implementation and experimental results.

This dissertation concludes in Chapter 6 with a discussion on the contributions and directions for future research.

Chapter 2

Background And Related Work

In this chapter, we briefly introduce techniques that are used or related to our multi-tenant key-value store system in a tiered memory architecture.

First, we introduce MRC and LRU replacement policy. Second, the random sampling replacement and K-LRU policy are discussed. Following we illustrate a spatial sampling technique used to lower MRC construction overhead. Then we analyze memory partition schemes in recent literature. Finally, we review the techniques employed in tiered memory management.

2.1 Miss Ratio Curve (MRC)

The Miss Ratio Curve (MRC) is a function mapping from cache sizes to miss ratios, given the MRC of a workload, one can immediately know the miss ratio for any cache allocation. Because of its capability in identifying workload locality and pattern changes, MRC has been a useful tool in cache memory management [4, 7, 8, 9, 10, 11, 12]. Figure 2.1 is an example MRC of MSR Web workload.

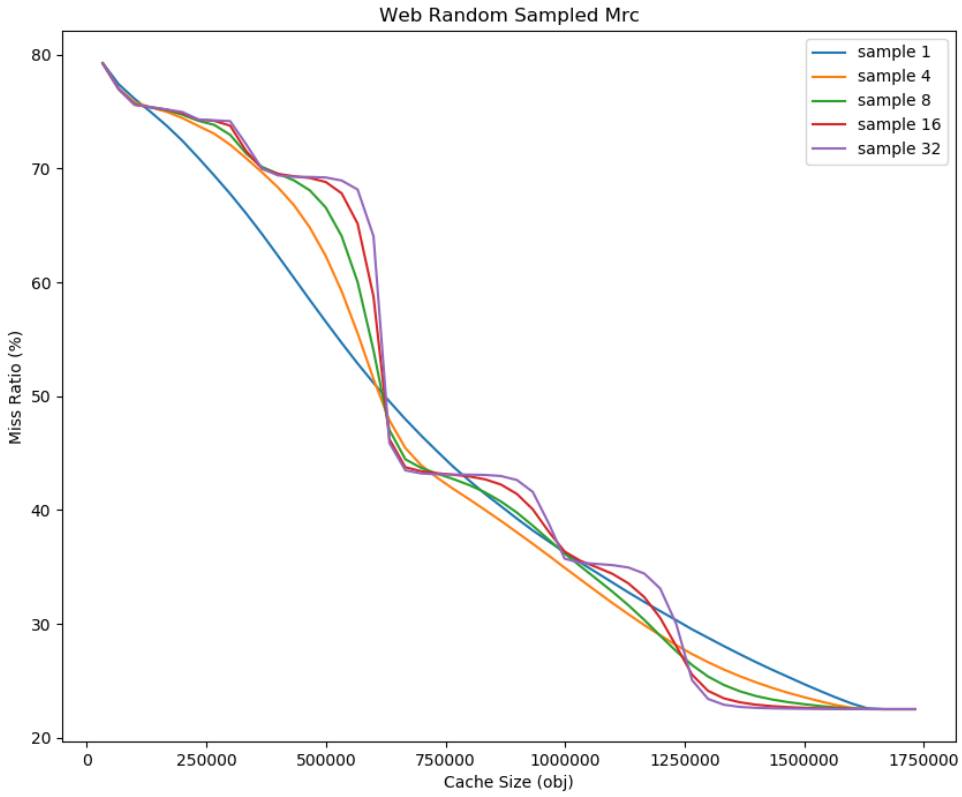


Figure 2.1: MSR web MRCs.

Since accesses in real-world commercial key-value stores show high data locality [3, 4], the Least Recently Used (LRU) replacement policy becomes a good choice as it can exploit the locality well. As of today, most studies on efficient MRC construction are focused on the LRU cache [7, 9, 13, 14, 15].

2.2 Random Sampling Replacement

Exact LRU implementation can be costly. In software caches, prioritizing items according to their last-access-time usually relies on linked structures to book-keep their orderings [5], and item evictions require list operations including pointer updates. All of these introduce space overhead and computation overhead. In addition, each time when an item is accessed, the LRU list must be locked to facilitate the update of corresponding LRU priority, resulting in extra performance degradation [6]. Memcached, another popular key-value store, only maintains the LRU structure at the slab class level [16].

In addition to the LRU policy, there are several sophisticated replacement policies such as ARC [17], MultiQueue [18], CACHEUS [19], and the Segment LRU recently implemented by Memcached [16]. While these algorithms generally perform effectively across various workloads, they share some common drawbacks. Firstly, these advanced policies necessitate the use of additional sorted data structures to keep

track of the relative order of objects, requiring the cache to allocate extra time and resources to maintain this order. Secondly, the data structures employed are typically quite inflexible; once a replacement policy is established, it becomes challenging to adjust the rules dynamically due to the inherent characteristics of these structures.

To avoid using expensive ordered-data structures and to improve performance, many existing schemes have adopted the idea of random sampling: On eviction, the cache randomly selects a small number of items and then evicts the item with the lowest priority. Ideally, the evicted item from a set of relatively small random sampled items could closely approximate the lowest priority in the whole cache [5].

2.2.1 K-LRU Policy

Redis, a widely-used commercial in-memory cache, employs an approximated LRU policy [20] that simplifies management by only tracking the access time for each object. During an eviction, it selects the candidate with the oldest access time from a randomly sampled subset of keys. Experiments have shown that even with a small sample size, this random sampling-based LRU closely mirrors the performance of exact LRU. We refer to this method as K-LRU, where K denotes the sampling size. Additionally, two innovative function-based cache replacement strategies, Hyperbolic caching for Redis [5] and LHD for Memcached [21], leverage this random sampling

approach to reduce the significant overhead associated with maintaining a complete ranking of all objects.

In Redis' random sampling-based LRU, each time when an eviction is needed, K keys are randomly sampled from all keys in the memory and added to an eviction pool. All keys in the eviction pool are sorted by their last access time and the one with the oldest time is evicted. The default setting of Redis is $K = 5$. Each time 5 randomly sampled keys are added to the pool for eviction decision. The number of sampled keys, K, is configurable but is fixed across Redis execution for the current design.

We run a collection of real-world enterprise server traces from Microsoft Research Cambridge [22] on Redis to plot the MRCs. Figure 2.1 shows the MRCs of five different sampling size K for trace *web* where cache size is represented as the number of objects. We observe that the MRCs of K-LRU with $K = 16$ can closely approximate those of the real LRU. A larger K normally indicates a closer behavior of eviction to the exact LRU policy, since the more keys are sampled to add into the eviction pool, the more likely the evicted object is near the least-recently-used side of the exact LRU list.

2.2.2 Impacts of Sample Size K

We use a simple trace example to illustrate the impact of sample size K over miss ratio by choosing three settings of K, where $K = 16$ simulates the real LRU, $K = 2$ represents a middle ground, and $K = 1$ means random eviction. Tables 2.1 to 2.3 show the cache content and eviction selection for 16-LRU, 2-LRU, and 1-LRU, respectively. Assume that cache capacity is 5. An evicted key is the one with largest access time chosen from K sampled keys, which are sorted based on their recent access time from top to bottom with the most recent item on top in the tables. Under $K = 16$, every access is a miss, miss ratio is 100%. Under 2-LRU, there are 9 misses and the miss ratio is 75%. Under random eviction, i.e., 1-LRU, there are 7 misses and the miss ratio has decreased to 58%.

This example shows that the trace pattern, cache size, and sample size K of K-LRU all have impacts on the miss ratio of a key-value cache. As pointed out by Jaleel et al [23], LRU is not able to explore well the reuse distances that are larger than the cache size. Note that the reuse distance between an access and its next reuse is the number of distinct accesses in between. In the sample trace, the reuse distance for all reuses is 6, which is bigger than the cache size 5. In this case, 16-LRU, which behaves close to the real LRU, fails to generate any hits. On the other hand, random eviction leads to most hits. In reality, we can observe a mix of short and long reuse intervals

Table 2.1
Redis Replacement: 16-LRU

Trace	a	b	c	d	e	f	a	b	c	d	e	f
Cached Keys	a	ab	abc	abcd	abcde	bcdef	cdefa	defab	efabc	fabcd	abcde	bcdef
Evicted Key						a	b	c	d	e	f	a
Sampled Keys						e	f	a	b	c	d	e
						d	e	f	a	b	c	d
						c	d	e	f	a	b	c
						b	c	d	e	f	a	b
						a	b	c	d	e	f	a
Hit	0	0	0	0	0	0	0	0	0	0	0	0

Table 2.2
Redis Replacement: 2-LRU

Trace	a	b	c	d	e	f	a	b	c	d	e	f
Cached Keys	a	ab	abc	abcd	abcde	acdef	cdefa	cefab	efabc	eabcd	abcde	acdef
Evicted Key						b		d		f		b
Sampled Keys						e		a		c		e
						b		d		f		b
Hit	0	0	0	0	0	0	1	0	1	0	1	0

Table 2.3
Redis Replacement: 1-LRU

Trace	a	b	c	d	e	f	a	b	c	d	e	f
Cached Keys	a	ab	abc	abcd	abcde	abdef	bdefa	defab	defac	efacd	facde	acdef
Evicted Key						c			b			
Sampled Keys						c			b			
Hit	0	0	0	0	0	0	1	1	0	1	1	1

in different phases. Based on this observation, we propose an approach to choosing K dynamically.

2.3 MRC Modeling

2.3.1 Stack Algorithms

Mattson's Stack Algorithm [24] is a generalized algorithm that models a general class of replacement policies. A replacement algorithm is called a stack algorithm if such replacement algorithm satisfies the inclusion property, that is, $B_t(C) \subset B_t(C+1)$, where $B_t(C)$ is a set of distinct objects in a cache of arbitrary size C at given time t. The inclusion property of the stack algorithm makes it possible to generate an MRC in just one pass of the trace. The algorithm models the cache as a stack, and the stack location i (stack top location = 1), where the referenced object resides, is called the object's *stack distance* (to the stack top). Under the stack model, an MRC can be calculated based on *stack distance distribution* (SDH): the miss ratio of a cache size c is the probability of stack distance greater than c.

Mattson's stack model [24, 25] was originally designed to simulate a range of replacement algorithms based on the assumption that all cached objects are of a fixed size. This assumption is suitable for hardware caches where each cache block is uniformly

sized. However, this fixed-size assumption does not necessarily apply to software caches. Recent research [26, 27] highlights that objects in in-memory caches can vary significantly in size, and the distribution of these sizes tends to change dynamically over time. Furthermore, Pan *et al.* [28] have shown that miss ratio curves, which are constructed under the assumption of uniform object sizes, can differ markedly from actual miss ratios when the workload exhibits a non-uniform size distribution.

Yang et al. [2] present a new probabilistic stack algorithm named KRR which can be used to accurately model random sampling based-LRU under arbitrary sampling size K. To handle variable object size in modeling K-LRU policy, Yang et al. propose a solution to add an additional array structure, *sizeArray*. Each entry of the *sizeArray* maintains a partial accumulation of stack size, specifically, the entry i of the *sizeArray* stores the total size of objects from stack top to stack position b^i, where b is the base parameter. Since the length of *sizeArray* is logarithmically bounded with respect to KRR stack length, the cost of maintaining the *sizeArray* is at most $O(\log M)$, where M is the stack size. With aids from *sizeArray*, they can make better estimations on byte-level stack distance. It is feasible to construct MRC of a variable object-size K-LRU cache for any K.

2.3.2 Spatial Sampling

The problem with the original stack algorithm is that it is very expensive, in both space and time, to obtain the actual SDH for a long trace because the asymptotic space/time cost of the stack algorithm is correlated with the number of unique references in the workload, which can be very large. Due to the large overhead, it is impractical to directly use a stack algorithm online. In order to make it suitable for online usage, the uniformly random spatial sampling described in SHARDS [7] becomes a widely adopted technique. Instead of feeding entire reference streams to the stack model, the spatial sampling technique uses the sampling condition $hash(L)\ mod\ P < T$, with referenced key L, modulus P, and threshold T, to collect only a subset of references. The effective sampling rate is $R = T/P$. As shown by Waldspurger *et al.* [7], for the majority of workloads tested, the sampled subset has very high statistical similarity compared to the original workload, even with R = 0.001. The spatial sampling technique can thus significantly reduce the number of tracked references for online MRC prediction.

2.4 Memory Partitioning

In real-world cache deployments, one cache instance usually serves multiple tenants, forming a multi-tenant environment. There are two basic memory partitioning strategies, equal partitioning, and free competition. In equal partitioning, each of the n tenants running simultaneously takes *1/n* of the total memory. This strategy seems to be fair to all tenants. However, due to access pattern change or trace locality difference, cache performance might deteriorate by offering some tenants more memory than needed, while others suffer from memory shortage. Free competition, on the other hand, is a first-come-first-serve policy, all tenants are competing for shared memory. The memory usage of tenants is decided according to various factors, including access rate, locality, object size, miss latency, etc. This is the default strategy that Redis employs. A tenant's throughput may be significantly affected by some noisy neighbors. To maximize the memory utilization of high-throughput multi-tenant storage systems, recent studies consider better memory partitioning schemes for applications based on online MRC construction techniques.

LAMA [4] is a locality-aware slab class level memory allocation for Memcached using the footprint theory [14] to model slab class trace locality and construct an MRC for each slab class. LAMA optimizes overall performance for all size classes, either total miss ratio or average response time. It can also be used in QoS-guaranteed

applications use cases.

Dynacache [29] targets improving hit rate of web applications, uses a bucketing scheme to estimate item stack distance in trace profiling, and allocates slab memory of Memcached for tenants.

Cliffhanger [30] employs a hit rate gradient estimation mechanism using shadow queue structures and incrementally transfers memory resources to the application that would benefit most from those that benefit the least.

Memshare [31] is a DRAM key-value cache memory partitioning system that optimizes the overall hit rate of applications and allows each application to specify its own eviction policy. It extends the Cliffhanger model to track a hit gradient for each application. Memshare abandons slab classes and introduces a segmented in-memory log to store application items. Varied-size items from different applications can be allocated the same segment and thus memory reallocation can be at item level. Memshare reserves a specified minimum amount of memory for each application to provide performance guarantees, and the remaining memory is allocated to maximize hit rate.

Both pRedis [32] and mPart [33] adopt AET [34] for online MRC construction and use dynamic programming algorithms guided by tenant MRCs to allocate memory.

I-PLRU [35] achieves minimized misses for a multi-flow LRU cache with an insertion-based pooled LRU paradigm. The cache space is pooled to serve multiple data flows but organized as a single list. Each tenant data flow is assigned an insertion position of the list. By configuring the insertion point dynamically, it proves that I-PLRU can reach the same minimum miss ratio of separated LRU caching. Robinhood [36] re-purposes existing caches to mitigate backend latency variability. Rather than solely caching popular data, it dynamically reallocates cache resources from "cache-rich" backends, which do not significantly impact request tail latency, to the "cache-poor" backends, thereby increasing hit ratios, reducing backend queries, and easing resource congestion, which all contribute to improved P99 request latency.

2.5 Tiered Memory Management (TMM)

As semiconductor manufacturing technology continues to advance rapidly, DRAM memory cells are shrinking to sizes where holding a reliable, detectable charge level becomes challenging [37]. Additionally, the cost per GB of DRAM is unlikely to decrease due to limitations in production process miniaturization. Concurrently, the growing demand for memory presents significant challenges to the total cost of ownership in cloud computing environments. A promising strategy to increase memory capacity while reducing cost is the incorporation of cheaper, slower memory tiers

for storing infrequently accessed or 'cold' data. Options for such memory technologies include fast flash memory [38], phase-change memory [39], Memristor RAM [40], Spin-transfer Torque RAM [41], and 3D XPoint [42].

In the following sections, we first introduce two broad categories for implementing TMM. Then we analyze the techniques used in TMM. At the end of this chapter, we briefly introduce several advancements in TMM.

2.5.1 Hardware-based TMM

Currently, the Linux system lacks efficient support for tiered memory, leading memory vendors to offer tiered memory solutions at the hardware level [43]. These hardware-managed memory tiers operate independently of the operating system's memory management mechanisms. This approach offers several advantages. Firstly, it eliminates the need for operating system support, as the hardware autonomously manages the memory tiers. Secondly, it involves very low overhead; for instance, the Memory Mode of Intel's Optane DC manages memory at the granularity of cache lines without the need to handle page tables or maintain the Translation Lookaside Buffer. However, this method also presents some limitations. Hardware systems have restricted insight into the actual memory usage patterns of applications, limiting them to employing simple, hardware-implementable techniques. For example, Intel's Optane DC

Memory Mode essentially uses DRAM as a direct-mapped cache for Non-Volatile Memory [44].

2.5.2 Software-based TMM

The limitations of hardware-based TMM have spurred the development of software-based TMM solutions [45, 46, 47]. In these systems, separate ranges of DRAM and slower memory types are distinctly managed, with a software layer—often the operating system or a dedicated library—responsible for data placement between DRAM and slower memory tiers. This software-driven approach provides greater visibility into how applications utilize memory and can implement more sophisticated management policies than those possible with hardware solutions. Most contemporary software-based TMMs leverage enhancements within the operating system, as current Linux OS implementations lack efficient mechanisms for migrating data between memory tiers, apart from traditional swapping to disk methods [48].

2.5.3 Memory Usage Profiling

Efficient Tiered Memory Management relies on the system's ability to accurately identify frequently accessed, or "hot" data and dynamically allocate it to the highest-performing memory tier available. This process ensures optimal use of faster, more expensive memory resources by prioritizing data that benefit most from reduced access times, thereby enhancing overall system performance and efficiency.

Most previous research applies OS-level memory usage tracking, including page table scanning, dirty bits, and TLB counting [49, 50, 51, 52]. Other research adopts hardware-specific profiling techniques, either based on AMD CPU's IBS and LWP, Intel CPU's PEBS [48, 53, 54] or Intel's PML for Virtual Machines (VM) [55].

However, using those techniques to determine which parts of applications are being accessed either does not scale with large address spaces (such as terabytes of memory) or can't be deployed as hardware transparent.

For cloud applications, particularly in-memory k-v stores like Redis and Memcached, workloads often exhibit memory footprints with large variants and dynamic access pattern changes. To effectively manage these, we propose the implementation of application-level memory usage tracking, and utilize a recency based priority for each data item to identify and categorize "hot" data. By focusing on application-level

data item priorities, this approach allows for more effective memory tier allocations, aligning memory usage more closely with actual application needs and performance requirements.

2.5.4 Memory Migration Techniques

Memory migration significantly affects system performance, particularly in cloud computing environments with multiple memory tiers. In these systems, substantial data volumes are often transferred between memory levels due to changes in memory usage patterns. Migration processes can be managed by the operating system kernel, which employs a page copy method that locks pages to ensure data consistency. While this prevents data from being altered during migration, it adversely impacts system performance due to the locking mechanism [55, 56, 57]. Additionally, updating Page Table Entries (PTEs) for moved pages can lead to costly context switches and Translation Lookaside Buffer (TLB) shootdowns [58, 59]. Alternatively, memory migration can be handled by a Direct Memory Access (DMA) engine [60], which operates concurrently with CPU tasks without utilizing CPU resources. However, the DMA operates at a lower frequency than the CPU and its initialization time is significant.

2.5.5 Compute Express Link (CXL) Memory Sharing

Compute Express Link (CXL) [61] is an open industry-standard interconnect designed to facilitate high-bandwidth and low-latency communication between host processors and various devices, including accelerators, memory buffers, and intelligent I/O devices. This interconnect is tailored to manage the increasing demands of high-performance computing tasks. CXL enhances heterogeneous processing and memory systems, crucial for sectors such as cloud infrastructure, artificial intelligence, machine learning, and analytics, by integrating coherency and memory semantics with existing PCIe-based I/O.

Two features of CXL are memory disaggregation and composability [61, 62]. Memory disaggregation is the ability to distribute memory across different devices, while still allowing multiple servers to share and maintain coherence. This approach eliminates the concentration of memory on a single device or server, provides more flexibility in designing the memory hierarchy and enables the use of different memory technologies with varying characteristics (e.g., capacity, bandwidth, latency).

Composability refers to the capability to assign disaggregated memory to designated CPUs or TPUs as needed, which considerably enhances memory utilization. Memory connected via Compute Express Link (CXL-Memory) is recognized by the system

as a NUMA node without an associated CPU, possessing unique memory attributes such as technology, generation, capacity, and bandwidth, distinct from the memory directly connected to the CPU [63].

CXL 2.0 and CXL 3.0 incorporate switching technology that allows a host to utilize memory from a pool comprising one or more devices. In CXL 2.0, it's possible for hosts to share only the resources, not the contents of the memory, with each memory region limited to a single coherency domain. On the other hand, CXL 3.0 introduces the capability of memory sharing. This feature enables the coherent sharing of CXL-attached memory across multiple hosts via hardware coherency. Consequently, multiple hosts can access the same memory region concurrently, ensuring that all involved hosts can access the latest data without the need for software-managed coordination.

CXL facilitates byte-addressable memory within a unified physical address space, allowing for straightforward memory allocation through standard memory allocation APIs [62, 64]. It offers access at cache-line granularity to connected devices while maintaining coherency and consistency via the underlying hardware. The bandwidth of the CPU-CXL interconnect parallels that of cross-socket interconnects found in dual-socket systems. Additionally, the latency associated with accessing CXL-Memory is comparable to that experienced with Non-Uniform Memory Access (NUMA) systems. This similarity to NUMA, coupled with the access semantics akin

to main memory, makes CXL-Memory a strong contender for the slower tier in data center memory architectures.

2.5.6 State-of-the-Art TMM Designs

HeMem [48] is a tiered main memory management system designed for non-volatile memory and big data applications. It uses asynchronous memory access sampling via CPU events, PEBS, to track memory access patterns, rather than page tables, enhancing its scalability to manage terabytes of memory efficiently. Asynchronous memory migration is also applied to minimize CPU overhead. It recognizes that certain ephemeral data structures do not scale unbounded in size, so they can be kept in the faster DRAM, rather than managing all data in tiered memory, to lower management overhead. HeMem considers the asymmetric bandwidth characteristics of NVM and places write-heavy data in DRAM to mitigate the impact and improve overall system performance.

vTMM [55] is a specialized tiered memory management system developed for virtualization environments. It aims to tackle the growing memory requirements of virtual machines and the limitations of conventional DRAM-only memory systems in terms of capacity and power consumption. The system enhances performance by automatically migrating pages between fast and slow memory based on their hotness.

It addresses virtualization challenges such as ensuring performance isolation, minimizing context switching, and supporting resource overcommitment. vTMM utilizes hardware-assisted page-modification logging (PML) for efficient page tracking and implements a multi-level queue design to minimize page tracking overhead. Pages are classified based on a weighted "page-degree" metric, which combines read and write frequencies to distinguish between hot and cold pages using a bucket-sorting algorithm. The system performs parallel page migration with minimal access interruption by leveraging the Page-Modification Logging feature and dynamically partitions memory among concurrently running VMs using a memory pool mechanism to enhance overall system performance.

TPP [63] presents an operating system-level mechanism that transparently allocates pages within CXL-enabled tiered memory architectures, which is critical for supporting the growing memory requirements of hyperscale applications. Using a user-friendly tool named Chameleon in Meta's server environments, TPP evaluates memory utilization patterns across a diverse array of datacenter applications. Chameleon leverages PEBS to monitor hardware-level performance metrics related to memory usage and produces detailed analyses of memory access patterns, covering both virtual and physical dimensions. The findings indicate that a considerable volume of memory accesses are to data that remain inactive for extended periods, suggesting a strategic possibility to relocate these less active pages to slower memory layers. TPP facilitates the strategic relegation of pages from primary memory to CXL-based

secondary memory, which accommodates incoming page requests while ensuring that critical pages currently in slower memory are quickly escalated back to faster primary memory.

TMTS [65] is a memory tiering management system designed by Google for optimizing memory access across different tiers seamlessly in large-scale computing environments. It introduces two crucial metrics at the machine level, named Secondary Tier Access Ratio (STAR) and Secondary Tier Residency Ratio (STRR), which serve to harmonize the advanced performance objectives and usage efficiencies across extensive application fleets, and to assess the effectiveness of the memory tiering implementation. TMTS employs a dual method to differentiate between frequently and infrequently accessed data, labeling data as "cold" if it hasn't been accessed within a predetermined timeframe. This system utilizes a cold age histogram to monitor the distribution of time intervals between accesses, facilitating the policy engine in recognizing patterns of application access and fine-tuning demotion settings as needed. For detecting frequently accessed pages, TMTS integrates two complementary methods: sampling and scanning. It leverages event-triggered sampling focused on last-level cache miss events to pinpoint recently accessed memory addresses in secondary tier memory, rapidly pinpointing potential pages for promotion. Additionally, TMTS conducts regular proactive scans of page access bits (A-bits) to identify hot pages that might not be captured by sampling. To keep the STAR within an optimal range to minimize the impact on performance outliers, TMTS deploys a flexible policy known

as secondary tier access directed demotions, which adaptively modifies the demotion age based on ongoing conditions. The system faces challenges such as the overhead of address translation, interference issues, and complications related to page size, emphasizing the necessity for proactive demotion and swift promotion detection. TMTS underscores the intricate task of managing memory tiers effectively at a grand scale and illustrates the importance of adaptive, tiering-aware scheduling and management of large pages to alleviate performance degradation and minimize access disruptions.

Lee et al. [66] investigate how heterogeneous memory tiering can boost the efficiency of key-value caches, specifically through a study involving CXL-based memory enhancement. They present a CXL 2.0 memory expansion solution, which includes a prototype CXL memory expander along with the Heterogeneous Memory Software Development Kit. This solution provides two distinct architectural approaches for key-value caches: Transparent Tiering and Data Tiering. Transparent Tiering integrates DRAM and CXL memory into a unified tier, maintaining transparency from the application's perspective regarding the diverse memory configuration. On the other hand, Data Tiering is tailored to minimize index latency sensitivity by allocating metadata and indices to DRAM while placing key-values in CXL memory, or a combination of both, based on the bandwidth needs of the stored objects.

Pond [64] is a memory pooling system that aims to improve DRAM utilization and reduce costs in cloud platforms while meeting performance requirements. It addresses

a key challenge of memory stranding, where unallocated memory remains despite all cores being rented. Based on CXL enabled low-latency load/store accessto pooled memory, the system proposes small-pool designs by pooling memory across 8-16 sockets, which is found sufficient to gain significant benefits. It uses machine learning models to predict optimal memory allocation for virtual machines (VMs) to resemble same-NUMA-node memory performance. The authors also demonstrate the practicality and performance of Pond for cloud deployment, highlighting its potential to provide significant cost savings for large cloud providers.

Chapter 3

Dynamically Configuring LRU Replacement Policy in Redis

To reduce latency in accessing backend servers, modern web services use in-memory k-v stores like Memcached and Redis to cache frequently accessed objects. Due to limited memory size, these stores must manage cache replacement when the data footprint exceeds cache capacity. Memcached uses the least recently used (LRU) policy, while Redis employs K-LRU, an approximated LRU policy, to avoid the overhead of maintaining LRU lists. We observe that K-LRU closely approximates LRU when K is large, but different values of K can result in varying miss ratios.

In this chapter, we propose *DLRU: Dynamic LRU*, which explores the configuration of

K in K-LRU, in order to improve the overall system performance. DLRU reconfigures K along with Redis execution. We adopt a scaled-down cache simulator to track the miss ratios of different Ks on the fly with a low overhead [67]. We develop a cost model to balance between the benefit of low miss ratio and the overhead of random selection and sorting of K-LRU. Our experiment results show that DLRU can always match the performance of the best K, and improve the overall Redis throughput over the default 5-LRU by up to 32.5%.

3.1 DLRU System Design

As discussed in Section 2.2, Redis sets up K during initialization, and K is fixed unless a client manually switches it. The eviction process does not require that K be fixed. So the basic idea of DLRU is simple: reset K automatically on the fly. As shown in Figure 3.1, we simulate K-LRU under various K. We dedicate one miniature cache for each K. We use a penalty cost model to pick one that minimizes overall time latency and reconfigure Redis in real-time. The implementation consists of two parts residing in server initialization and command dispatcher, respectively. In server initialization, we set an interval size measured as the number of GET requests. Later, for every interval, DLRU will decide if a new K needs to be set. We also initialize the miniature caches in this stage. In the command dispatcher, once a GET key command is detected, DLRU determines whether to sample such key. If a key is

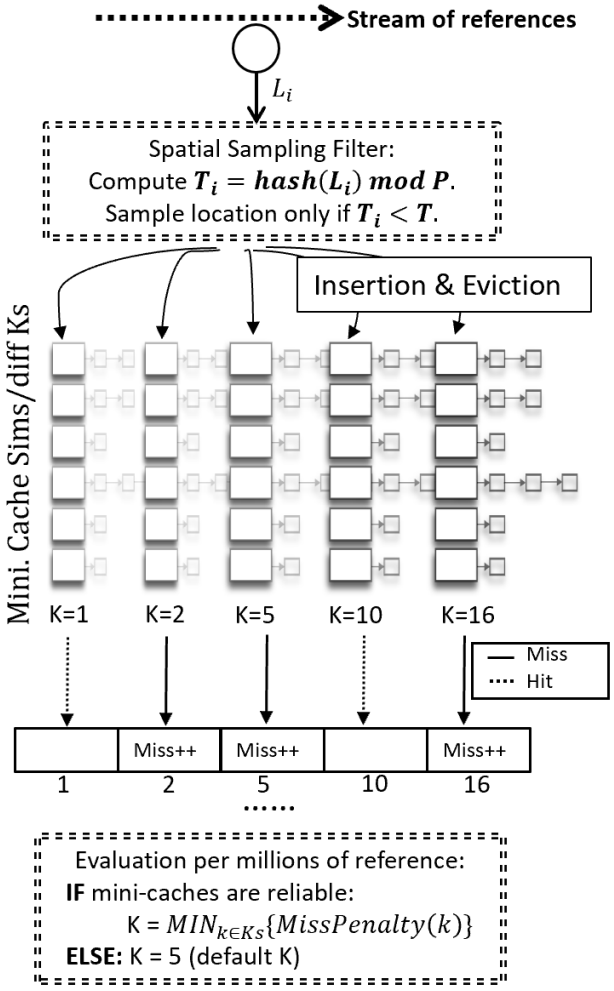


Figure 3.1: DLRU overview

sampled, it is fed into every miniature cache. After calculating and comparing the overall miss penalty for each K, the Redis `server.maxmemory_samples` parameter is set to be the optimal K with the least predicted penalty.

3.1.1 Miniature Cache Simulation

In order to make the selection of sample size K, the miss ratio of the corresponding K must be predicted in real time. We adopt a lightweight scaled-down approximation technique, the Miniature Cache Simulator, to reduce the overhead of capturing trace patterns and miss ratio tracking [67]. The miniature cache proposed by Waldspurger et al. simulates the actual cache by scaling down, by several orders of magnitude, both the original accesses and the cache size. It can accurately model the behavior of the original cache with any given eviction policy.

During the processing of an online trace, all references are hashed, and only when the hash value of a key is less than a threshold, that key of reference is sampled and stored (cached) into a hash table. Let the T and P be the threshold and modulus. The sampling condition for any referenced key L is $hash(L) \mod P < T$. The miniature cache sampling rate is thus $R = T/P$. This ensures all requests to the same key will be sampled. The small and spatially hashed samples of the requests show a statistical similarity against the whole references. Miss ratio can then be extracted from the miniature cache by counting the number of misses against the total sampled references. Experiments show that small values of $R = 0.01$ or even $R = 0.001$ can yield very accurate results. Such a low sampling rate implies low time and space overhead even for a long execution.

We place a filter in Redis to identify keys that need to be sampled. Modulus P is set to a power of two, and threshold T is fixed according to a given miniature cache sampling rate R as $T = P * R$. In a periodic interval window (default is 5 million GET requests), a randomly sampled small subset of references are selected to feed into the miniature caches.

As discussed in Section 2.2, $K = 16$ and $K = 1$ represent real LRU and random eviction, respectively. We add three other settings in between where $K = 2$, $K = 5$, and $K = 10$. Five independent K-LRU miniature caches, corresponding to K = 1, 2, 5, 10, 16, are fed with this subset of keys at the same time. Although more miniature caches of different Ks between 1 and 16 could be evaluated, the performance gap between a small interval of K-settings is not significant. In addition, more miniature caches will also introduce more simulation overhead.

Each miniature cache first looks up the sampled keys in its own key space, which is maintained in its own hash table. If the key does not exist, a miss occurs and the key is cached. A cache replacement using the corresponding K-LRU policy is invoked if the miniature cache is full. To determine if a miniature cache is full, the currently used cache size is compared with the maximum cache size. The currently used cache size is simply the number of items stored in the miniature cache.

The maximum cache size, in terms of the number of objects, is computed using Eq. 3.1 below, where the average item size is extracted directly from Redis statistic.

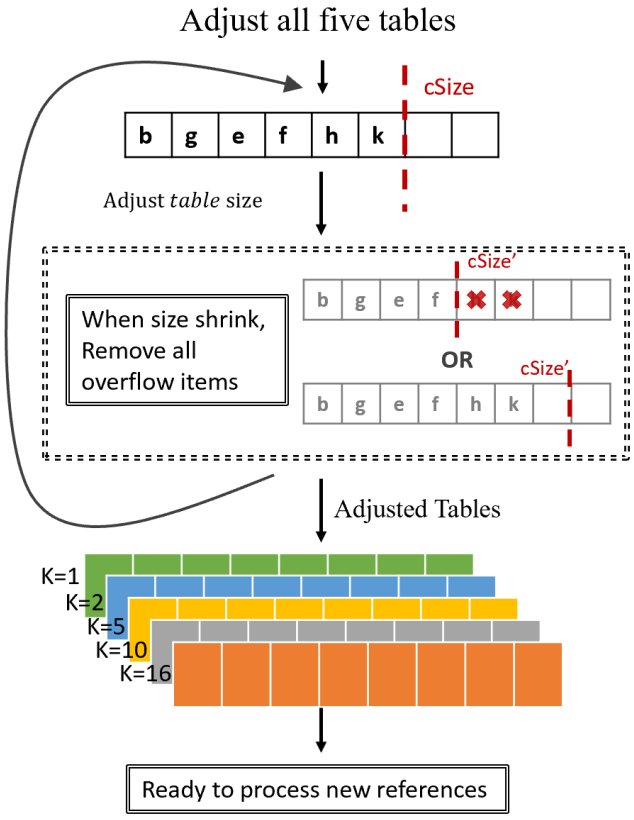


Figure 3.2: Hash table resizing

As the average item size can change across intervals, the $Max\ cSize$ should adjust accordingly. As shown in Figure 3.2, all miniature caches are required to adjust their size after every five million of requests. When the $Max\ cSize$ shrinks, we simply remove all overflow items in the miniature cache. Note that since the position of an item in the miniature cache is randomly determined, the removals of n items from the tail of the miniature cache are statistically equivalent to randomly removing n items.

$$\text{Max cSize} = \text{server.maxmemory} * R/\text{average item size} \tag{3.1}$$

Ideally, miniature caches should simulate K-LRU accurately. However, if during an interval, not enough distinct references are cached, the accuracy is not guaranteed [67]. We observe that when the actual number of sampled distinct references in a five million request interval is greater than 256, miniature simulation can deliver acceptable accuracy, which is also observed in the original work by Waldspurger et al [7]. In our implementation, we set R to 1/200, i.e., one of every 200 requests is sampled. This sampling rate works well for most cases. If in any evaluation interval, less than 256 distinct references are collected, we consider that the miniature cache is too small to predict a reliable miss rate. In this case, the default $K = 5$ is configured for the following interval. Note that 256 is a very small number compared to the total of five million requests. Only on a few occasions, DLRU will need to go back to the default.

3.1.2 Miss Latency and Eviction Process Overhead

To estimate the performance impact of misses, we need to know the miss latency. In this work, the miss latency is defined as the time interval from the miss of a GET operation in the key-value cache to the completion of a SET operation with the same

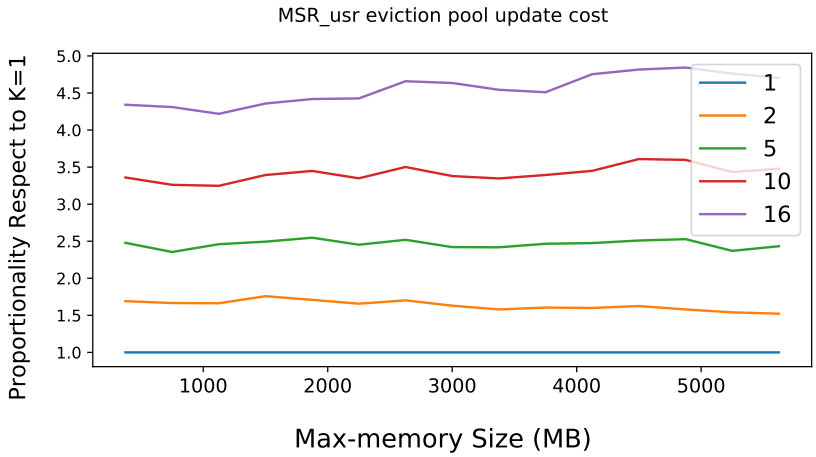


Figure 3.3: Overhead measurement

key sent by the client front end. We measure the miss latency on the fly for each interval of 5 million references. We track the miss penalties in the previous interval and use their mean as the miss latency for the DLRU decision in the next interval.

The cache eviction overhead comes from sampling and access-time comparison. The first one is the operation of randomly sampling the required number of K keys from all that in memory. The other is the operation of merging with the eviction pool and finding a key with the largest last access time for eviction. It is challenging to actually measure this overhead for all K-LRU settings in real time as in any time interval, only one setting of K can be measured.

In our experiments, we observe a proportional relationship of such overhead between different settings of K. Let OH_K be the mean sampling and comparison overhead for K-LRU. Figure 3.3 shows OH_K for $K = 1, 2, 5, 10, 16$ for `MSR-usr` on a fixed-K-configuration Redis server. The range of each normalized curve is relatively small. In

other words, if we measure the OH_K at a time window, we can estimate other $OH'_K s$ based on the pre-measured proportionality ratios.

3.1.3 DLRU Cost Model

In order to deal with access pattern changes, we divide the reference stream into fixed-size intervals according to the number of GET requests. In our experiments, the default interval size is set to 5 million GET accesses. The cost model in this section estimates the overall miss penalty for K-LRU in the current interval and use it to guide the choice of K for the next interval.

Let p be the average miss latency, OH_K be eviction overhead and M_K be the miss count gathered in the past interval using the miniature model. We estimate P_K the overall miss penalty for K-LRU as follows.

$$P_K = M_K * (p + OH_K) \tag{3.2}$$

Our goal is to choose a K with the minimal P_K. Then Redis server is reconfigured with the optimal K for the following interval. Normally, p is orders of magnitude greater than OH_K, so the impact of the latter over the overall miss penalty is trivial. Selection of K is dominated by the miss counts predicted by the miniature caches.

Our scheme will choose the K with the smallest miss ratio. However, we observe that, if the miss counts are very close to each other for different Ks in an interval, OH_K can become a deciding factor for the overall miss penalty. In this case, our scheme will prefer a smaller K.

3.2 Experimental Evaluation

In order to evaluate the effectiveness of DLRU, we first give a brief description of the experimental setup. Second, we evaluate the accuracy of the predicted miss ratio. Third, we compare the performance difference between Redis with default K and Redis with DLRU. Finally, we discuss both the time and space overhead of our selection scheme.

3.2.1 Experimental Setup

3.2.1.1 System Configuration

We use two separate machines for evaluation. Machine A is configured with Intel(R) XEON(R) E5-2620 v4 2.10GHz processor with 20 MB shared LLC and 128 GB of memory, and the operating system is Ubuntu 16.04.6 LTS with Linux kernel 4.4.0.

Machine B is configured with Intel(R) Xeon(R) Gold 5118 2.30GHz processor with 34 MB shared LLC and 188 GB of memory and the operating system is Fedora 31 with Linux kernel 5.6.13. All major evaluations are done on machine A, Machine B is only used in Section 3.2.4. We have implemented DLRU on top of Redis-4.0 [68] with the default Jemalloc allocator, and use *mutilate* [69] for request stream generation.

Initially, mutilate converts references in a workload to Redis GET commands, when Redis returns a miss, Mutilate will immediately follow a SET command. There is no back-end database in our setup, all KV pairs are generated from the mutilate client on the fly. With such a setup, the miss latency is simply the total setback time between mutilate and Redis. Additionally, both Redis and Mutilate are running on localhost. It yields relatively low access latency when compared to more typical cases where clients are run on a remote site. In a real system, the miss latency will be much higher. DLRU will still function as it measures the miss latency on the fly. With a higher miss latency, DLRU can only perform better as the overall miss penalty is higher.

3.2.1.2 Workloads

We use the MSR Cambridge storage workloads and their variants in our evaluation [22]. The original MSR suite contains traces from 13 different enterprise data

center servers. It covers a variety of access patterns, which is sufficient for us to evaluate the effectiveness of DLRU. We first evaluate the MSR traces under simplified conditions with uniform object size, where the object size of each key-value pair is 200 bytes. Next, we use the MSR traces' original object size to show that DLRU also improves the performance of Redis under general circumstances. The MSR suite contains a couple of small traces which only take Redis roughly 10 minutes to process entire request streams. In order to better visualize the improvement from DLRU, we repeatedly concatenate the same trace to coin a roughly one-hour-long request stream. For notation purposes, as an example, `src2_10` is generated by concatenating MSR's `src2` trace 10 times. Lastly, in Section 3.2.3.3, we merge multiple MSR traces with different access patterns to demonstrate how DLRU selects an optimal K when the access pattern changes.

3.2.2 Miss ratio

For DLRU to make meaningful decisions, the five miniature caches must correctly simulate Redis replacement patterns under different Ks. We compare the actual Redis miss ratio for every 5 million requests with the predicted miss ratio yielded by the miniature cache with the corresponding K. Figure 3.4 shows the miss ratios over time for all MSR traces.

To quantify the accuracy of the predicted miss ratio, we follow the error metric used in [7], the mean absolute error (MAE). We calculate the MAE for each trace in Figure 3.4, and the average MAE across all traces is 0.031. We notice that there is an obvious vertical shift between predicted and actual miss ratio for the workloads with relatively small working set size such as `stg_16`. In practice, we find that the shift is consistent for all Ks and it is not a problem of DLRU decision. We attribute the shift to the bias introduced by spatial sampling of miniature modeling. Since all five miniature caches use the same subset of keys from spatial sampling, they are likely to suffer from the same relative vertical shift.

In Figure 3.5, we use a synthetic workload that contains two separate phases. One phase is designed with poor temporal reuse, where random evictions are preferable, and the other phase is designed with high temporal reuse, where evicting the LRU objects is preferable. Figure 3.5 contains the miss ratio predicted by DLRU and Redis miss ratio. We also plot miss ratios for both miniature caches with $K = 1$ and 16 to illustrate that DLRU always selects the optimal K over time. Note that in the initial phase where the miss ratio of 1-LRU and 16-LRU are roughly the same, DLRU chooses $K = 1$ (the choice of DLRU is shown in the square boxes). This is the case when the eviction overhead decides the K selection as a smaller K implies a lower overhead (OH_K in Eq. 3.2).

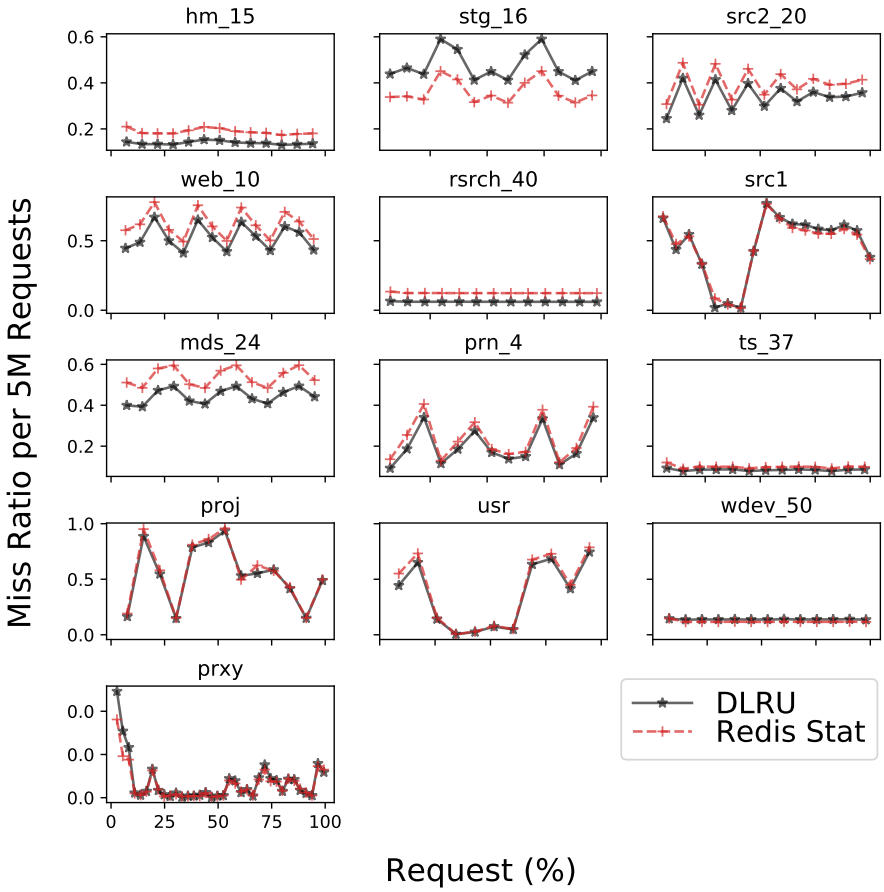


Figure 3.4: Miss ratio prediction accuracy

3.2.3 Overall Throughput

In order to measure the performance gain of our model, we employ throughput as the evaluation metric. Since all workloads we use are fixed-length traces, throughput is the ratio of the total number of requests to the overall execution time. In this section,

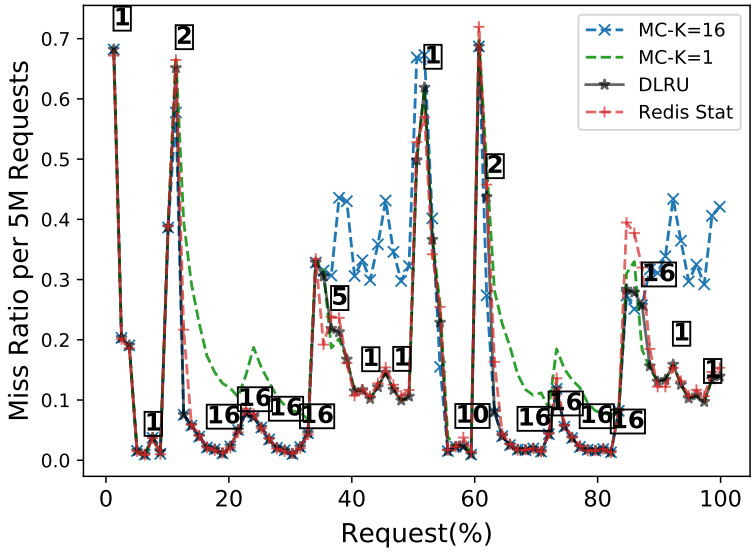


Figure 3.5: Miss ratio prediction for a phase-changing workload

we compare DLRU, 1-LRU, and 16-LRU, with Redis default sample size $K = 5$ (5-LRU). In practice, the 16-LRU behaves almost identically to the true LRU, and the 1-LRU is basically the random replacement policy. We will demonstrate the benefit of DLRU that exploits the access pattern of the current request stream on the fly.

3.2.3.1 Uniformly-Sized MSR Workloads

In this set of experiments, we set the item size uniformly to 200 bytes for all MSR workloads. We divide the 13 MSR workloads into two separate sets, A and B. Set A includes those MSR workloads that have notable difference in terms of miss ratio

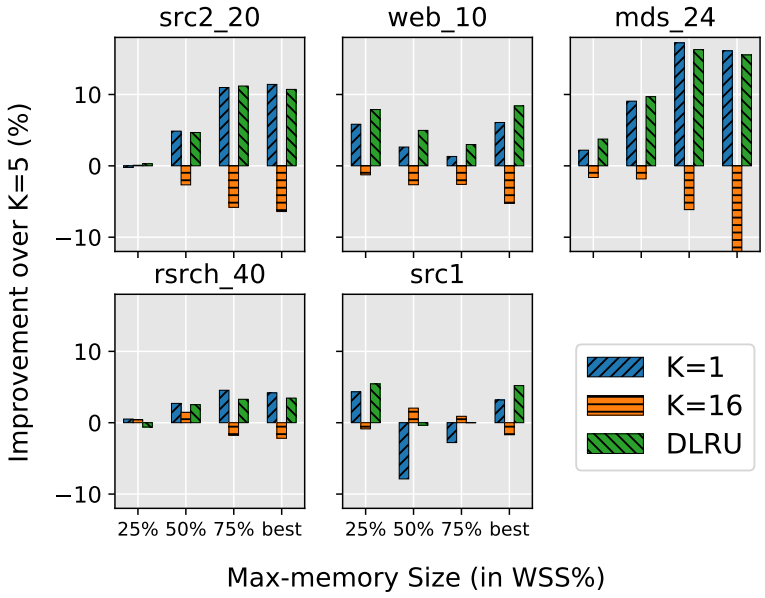


Figure 3.6: Throughput improvement with respect to 5-LRU for uniform item size. Set A (best: memory size that yields the largest difference in terms of miss ratio)

under various Ks (1, 2, 5, 10, 16). Many workloads in set A consist of long-stream repeated patterns which are in favor of random replacement when Redis' max-memory is smaller than their working set sizes (WSSs). Set B consists of the MSR workloads that have relatively small differences in terms of miss ratio under various K. Figure 3.6 and Figure 3.7 show results from 5 representative MSR workloads in set A and B, respectively. To evaluate the performance of DLRU under different Redis' `max-memory`, the Redis `max-memory` is set to 25%, 50%, and 75% of the working set size of the evaluated workload. The "best" in Figure 3.6 is the memory size where there is the largest gap in miss ratio between $K = 1$ and $K = 16$.

In set A, compared to the default sample size K=5, DLRU increases throughput by

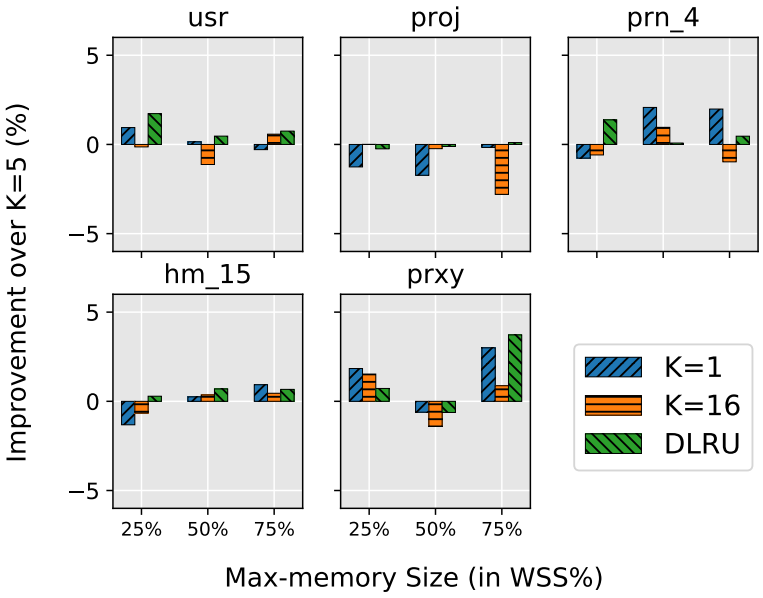


Figure 3.7: Throughput improvement with respect to 5-LRU for uniform item size. Set B

as much as 16.3%. DLRU matches or outperforms 5-LRU in all benchmarks and all max-memory settings. It is worth noting that when max-memory is set to 25% of WSS, the `src1` workload shows favor to random replacement ($K = 1$). We see 4% and 5.5% improvement for $K = 1$ and DLRU, respectively. Then we see an 8% degradation for random replacement when max-memory is set to 50% of WSS. When the max-memory is set to 50% of WSS, Redis is able to keep all hot items, random replacement is no longer the favorite choice. DLRU's auto selection of K is able to perform the best in both cases.

In set B, as shown in Figure 3.7, the largest improvement by DLRU is 3.7% from `prxy`, which is modest compared to the workloads in set A. Set B consists of workloads that are insensitive to change in K, i.e., all workloads perform mostly the same

under random replacement or LRU replacement, which results in limited room for improvement under DLRU. But on the upside, we still see that DLRU increases the throughput of all workloads by 1%, on average, compared to 5-LRU. In set B, both random replacement ($K = 1$) and 16-LRU ($K = 16$) perform nearly identical to default $K = 5$, with a difference of 0.3% and -0.2%, on average, respectively.

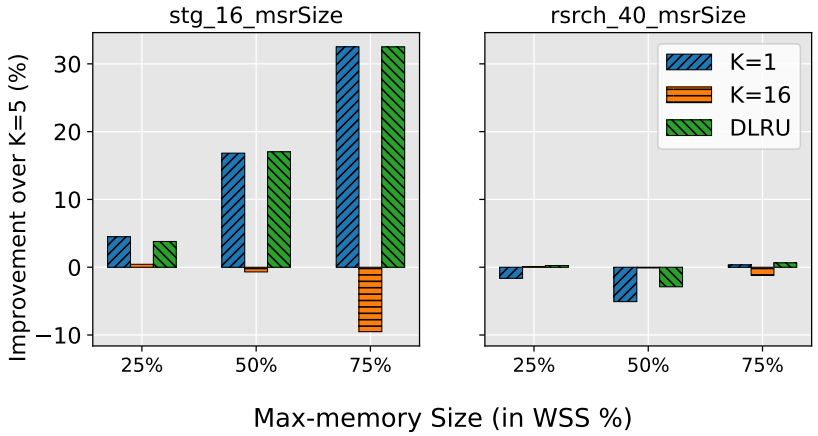


Figure 3.8: Throughput improvement with respect to 5-LRU for nonuniform item size

3.2.3.2 Non-Uniformly-Sized MSR Workloads

Next, we evaluate the performance of DLRU under non-uniformly-sized items. Figure 3.8 shows the results from two representative MSR workloads, where the size of each item is directly adopted from the original MSR traces. As the increased item size increases the miss penalty, we observe better improvement in some workloads. For `stg_16`, DLRU increases the throughput by 32.5% compared to default $K = 5$ at the memory size of 75% WSS. In `stg_16`, the item size distributions are relatively

stable across DLRU intervals. Our model, which uses average item size for cache simulation, works well. However, for `rsrch_40`, the average item size fluctuates across the request stream, which hurts the miniature cache accuracy. Despite this drawback, DLRU still shows the best performance at 25% and 75% WSSs, while there is a slight degradation compared to the default at 50% WSS.

3.2.3.3 Synthetic Two-Phase Workload

We evaluate the performance of the two-phase workload discussed in Section 3.2.2. Note that the workload consists of phases favoring random replacement and phases favoring LRU. As shown in Figure 3.5, a static choice of K would fail to make the best out of both phases. Figure 3.9 shows the improvement from DLRU when Redis' maxmemory is set to 30% of the WSS. The overall throughput is improved by 6.4% when compare against default $K = 5$.

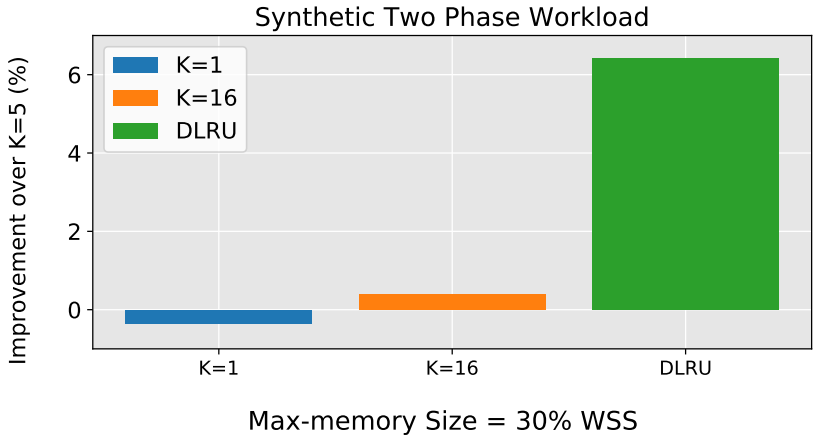


Figure 3.9: DLRU improvement on a two-phase workload

3.2.4 Sensitivity

As mentioned in Section 3.1.2, we observe a proportional relationship in the cost of updating the eviction pool under different settings of K. To verify that such observation is consistent over different machines, we collect the cost on both machine A and machine B (See Section 3.2.1.1) with Redis set to various max-memory sizes (10MB - 9GB). Table 3.1 shows the mean constant of proportion ratios with respect to $K = 1$ and their standard deviation over various max-memory sizes. The standard deviation is low. The results from both machine A and machine B agree with our observation: The costs of eviction under different settings of K are relatively proportional.

Table 3.1
Ratio of eviction process cost in Redis under different settings of K

Machine A		
K	Ratio	SD
1	1.00	0.00
2	1.64	0.05
5	2.37	0.06
10	3.18	0.13
16	4.31	0.18

Machine B		
K	Ratio	SD
1	1.00	0.00
2	1.64	0.09
5	2.47	0.14
10	3.37	0.28
16	4.40	0.43

3.2.5 DLRU Overhead

3.2.5.1 Space Overhead

In our implementation, the space overhead is dominated by the five hash tables, which are used to simulate cache behavior under various K. When applying a fixed-rate version of the miniature cache, the size of the hash table will depend on both the sampling rate R and the average item size. Each item in the hash table, including auxiliary fields such as the hash handle, consumes 136 bytes. We can estimate the percentage of memory overhead relative to overall allocated Redis memory as follows: $136\ bytes\ *\ 5\ Tables\ *\ R\ /\ average\ size\ of\ KV\ pair$. As an example, the `stg_16` trace contains 1.6 million unique Key-Value pairs, the average size of each KV pair is 70KB and we set $R = 1/200$. In this case, the total additional space overhead introduced by DLRU is about 0.005% of the overall allocated Redis Memory.

3.2.5.2 Time Overhead

The time overhead of DLRU mostly comes from simulating miniature caches under various K. The miniature cache technique helps reduce time overhead drastically. We only sample one request for roughly every $1/R$ requests (one in every 200 in our

evaluation). For `stg` with an average key-value pair size of 70KB, we observe that the time overhead of DLRU is only 0.027% of total execution time, which is insignificant compared to the potential gain from DLRU. The time overhead for other workloads is similarly low.

3.3 Chapter Summary

This chapter presents a new replacement policy, DLRU, for Redis. DLRU is built upon the existing K-LRU policy. Rather than fixing K across Redis execution, DLRU chooses an optimal K in every execution interval based on a cost model that estimates the miss penalty. We engineer a dynamic system using a low-overhead cache simulator. Experimental results demonstrate that it works well for both simplified and general conditions regarding object size, and can always match the best K performance or outperform a fixed-K system across a range of storage traces. To our best knowledge, DLRU is the first system to dynamically select a replacement policy along with key-value cache execution to adapt to the access pattern changes.

Chapter 4

Memory Partitioning for Multi-Tenant K-V Store

In this chapter, we focus on the multi-tenant k-v store use case. Multiple applications/tenants share a single Redis cache instance, where the available memory is partitioned for each tenant to meet their caching requirements. Since memory space is limited, maximizing the utilization of the shared memory pool is critical for system performance. Additionally, the recent research on cache-sharing models that guide memory allocation among the tenants, including LAMA [4], mPart [33], pRedis [70], and Memshare [31], are all based on the exact LRU policy. To the best of our knowledge, the memory management for the multi-tenant k-v store that employs K-LRU still needs to be addressed.

To address those challenges, we introduce kRedis, a reference locality- and latency-aware memory partitioning scheme, to improve the performance of the in-memory multi-tenant key-value cache that utilizes random sampling-based replacement. kRedis guides the memory allocation among the tenants and dynamically customizes K to better exploit the locality of each individual tenant. Evaluation results over diverse workloads show that kRedis delivers up to a 50.2% average access latency reduction, and up to a 262.8% throughput improvement compared to Redis. Furthermore, by comparing with pRedis, a state-of-the-art design of memory allocation in Redis, kRedis shows up to 24.8% and 61.8% improvements in average access latency and throughput, respectively.

4.1 kRedis System Design

In existing MRC-guided partitioning designs [4, 33], reference keys are randomly sampled to construct reuse time histogram, from which an MRC for each application can be calculated using the footprint model or the AET model [9, 34]. At a specified interval, a dynamic programming algorithm is invoked to minimize the number of expected misses based on the constructed MRC. This partitioning scheme is based on the fact that the MRC of each tenant is fixed, which is true for Memcached when applying the exact LRU eviction policy. However, for Redis and other caches that employ K-LRU, the sampling size K's impact on miss ratio can be significant.

This means there are multiple MRCs available corresponding to different K for each application. Therefore, the search space for finding the optimal partition solution significantly increases.

We introduce a memory partitioning scheme for K-LRU Redis cache named *kRedis* assuming that K is configurable on the fly. To reduce the search space, kRedis dynamically allocates memory based on a merged MRC from a small set of MRCs of different Ks. The MRCs of each K-LRU cache for each tenant application are constructed online using the KRR model described in Section 2.3.1, and spatial sampling technique discussed in Section 2.3.2.

4.1.1 Merged MRC

Intuitively we need the MRCs with different Ks for each application, where we can choose the MRC_K that yields the minimum miss ratio at cache size C and record the corresponding sampling size K. In order to avoid expanding search space, we simply merge several MRC_Ks of each tenant into one MRC where only the lowest miss ratio and respective K at every cache size are recorded. For each tenant, miss ratio curves MRC_K with different K are merged into a single miss ratio curve $mMRC$ as following:

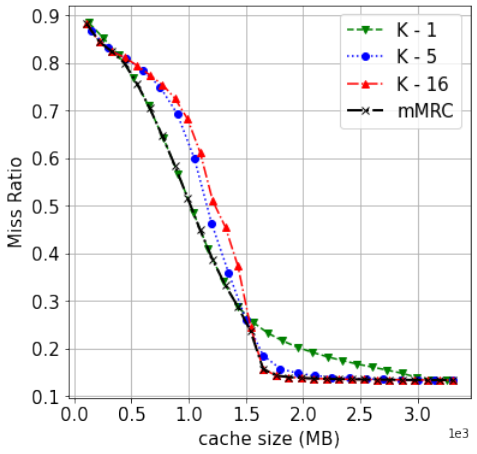


Figure 4.1: Merge K-LRU MRCs to $mMRC$.

$$mMRC(C) = min\ MRC_K(C) \tag{4.1}$$

Now the optimization problem can be reduced to a partitioning problem with each tenant having one fixed MRC, $mMRC(C)$. Figure 4.1 shows three K-LRU MRCs of MSR src1 with $K = 1$, 5, and 16 merged to one $mMRC$. The sampling size K is not shown on $mMRC$, it is encoded in the data structure of $mMRC$ so that later it can be used to configure the eviction policy of the corresponding tenant.

4.1.2 Memory Partitioning

Similar to mPart and pRedis [33, 70], a memory partitioning scheme is computed at the end of each periodic interval window, which is preset as the number of requests (default is 1 million). At each evaluation interval, for each tenant, we measure its

average miss latency and access rate, and construct the K-LRU MRCs by spatial-sampling its requests. We then determine the memory partitioning scheme using a dynamic programming algorithm. The partitioning is then enforced at the next interval. Specifically, there are four steps in each evaluation interval.

Step 1: Latency & Access Rate Measurement

We record the cumulative miss latency and the number of cache misses to calculate the average miss latency p for every application sharing a memory pool. The tenant access rate a is estimated as the rate of interval time and tenant access count in the interval window.

Step 2: Merged K-LRU MRC Construction

For each tenant, we construct MRC_K for several small Ks on the fly based on spatial sampling and the variable object size-aware KRR model, then derive the merged $mMRC$ for each application as discussed in Section 4.1.1.

Step 3: Memory Partitioning Scheme and K Selection

To take the impact of miss latency into account, we estimate interval miss latency P_i for application i as follows.

$$P_i = mMRC_i(C_i) * a_i * p_i \tag{4.2}$$

Our goal is to minimize the overall miss latency for a set of N tenant applications in a Redis cache instance with total memory M.

$$min \sum_{i=1}^{N} P_i = \sum_{i=1}^{N} mMRC_i(C_i) * a_i * p_i \tag{4.3}$$

$$subject\ to \sum_{i=1}^{N} C_i = M$$

Inspired by DCAPS [11], kRedis memory partitioning could achieve various optimization targets by adjusting Equation 4.3. For example, the following metric can be adopted to optimize hit throughput which is defined as the number of GET hits per access time.

$$max \sum_{i=1}^{N} TP_i = \sum_{i=1}^{N} (1 - mMRC_i(C_i)) * a_i \tag{4.4}$$

The optimization problem of Equation 4.3 can be solved using dynamic programming similar to mPart [33] and pRedis [70].

At the end of each evaluation interval, we run the memory allocation algorithm presented in Algorithm 1. We calculate the minimum overall miss latency and record the memory allocation for each application. The i loop (line 8) and j loop (line 9) combined find the best latency when the first i tenants are allocated j amount of memory. The innermost C loop (line 10) enumerates all possible allocations of tenant i subject to the upper bound of memory size j (line 10). The second loop nest from line 21 to 24 backtracks optimal memory partition recorded in $\{A\}$.

The time complexity of such dynamic programming is $O(VM^2)$, where V is the number of applications and M is the size of the memory pool. In real applications, the memory bound M could be a large value in bytes, but we use configurable larger granularity G in memory allocation, for instance, 1 MB or 10 MB, according to the application profiles. Then the time complexity becomes $O(V(M/G)^2)$ which is affordable for online usage.

Step 4: Dynamic Memory Allocation and K Adjustment

Once the memory partitioning scheme is determined, Redis memory should be allocated for each application accordingly. Inspired by the work of pRedis [70], we maintain two arrays to book-keep the amount of memory used in each tenant and the suggested memory allocation by our model. To ensure tenant references are processed under its appropriate K-LRU eviction policy, we configure the sampling size K_i for application i according to $mMRC_i$.

Algorithm 1 Memory Allocation

Require: M ▷ Total cache memory
Require: {V} ▷ Set of tenants
Require: {mMRC} ▷ Set of merged MRC for each tenant
Require: {a} ▷ Set of access rate for each tenant
Require: {p} ▷ Set of average miss latency for each tenant

```
1: procedure ARBITRATE
2:     for i ∈ V do
3:         for j ← 0 to M do
4:             f[i][j] ← ∞
5:         end for
6:     end for
7:     f[0][0] ← 0
8:     for i ∈ V do
9:         for j ← 0 to M do
10:            for C ← 0 to j do
11:                miss_i ← mMRC_i(C) * a_i
12:                latency ← f[i − 1][j − C] + miss_i * p_i
13:                if latency < f[i][j] then
14:                    f[i][j] ← latency
15:                    Target[i][j] ← C
16:                end if
17:            end for
18:        end for
19:    end for
20:    T ← M
21:    for i ← N; i → 1 do
22:        A_i ← Target[i][T]
23:        T ← T − Target[i][T]
24:    end for
25:    return {A}
26: end procedure
```

We adjust the tenant memory usage by modifying the Redis eviction process. Initially, Redis memory size is maintained by the eviction procedure named freeMemoryIfNeeded. Each time Redis receives a request, this procedure checks the used memory against the max-memory setting and free items if needed. In kRedis, we pick the application in which the used memory is greater than the suggested memory

for eviction. Thus the tenant space is adjusted on the object level and the pace of adjustment is dependent on the tenant's request pattern.

4.1.3 Efficient Random Sampling Eviction Design

The challenge in step 4 of Section 4.1.2 is that from the whole Redis key space, how can we effectively sample K keys that belong to a specific tenant. pRedis [70] adopts a bloom filter to determine the tenant belonging of each sampled key in the process of eviction. However, the bloom filter time overhead can be notable according to our evaluation. Each time a key is stored in the cache, the bloom filter needs to check if such key is a new key, then map the key to its tenant in the bloom filter structure. On evictions, every randomly sampled key must be checked to decide if it belongs to the memory-overusing tenant. According to our evaluation, when there are 4 tenants, the key space size of each tenant is about 12 million, and the cache max-memory is set to 50% of the working set size, the total time used in the eviction process is about 50 seconds, in which the bloom filter judgment time takes 15 seconds or 30%. On average, the bloom filter judgment takes 2 μs to identify a key for eviction. If the number of tenants and key space increase, the bloom filter time overhead in the K-LRU eviction process will only be larger.

Our solution is to separate key dictionaries for the tenants in Redis. The original

Redis design uses a single dictionary, where the keys of all the tenants reside, so there is no ready-made support for multi-tenant memory partitioning in a single Redis instance. By using multiple dictionaries, the K-LRU eviction is to simply sample K keys randomly in the desired tenant's dictionary and the bloom filer is no longer needed in both setting and eviction processes. In practice, it is trivial to distinguish tenant keys using the unique client ID embedded in the Redis data structure or other available parameters such as the source socket id of a request.

The data objects of Redis are stored as dictionary entries in the hash table and connected by pointers. Our multi-dictionary design only sets up multiple tenant-wise hash tables pointed by the dictionary header, which has a negligible effect on the Redis command processing, and the overhead is the metadata of the dictionary header, which is in a total of 176 bytes per tenant.

4.1.4 Implementation

Unlike the LRU stack, the KRR stack only shifts a small subset of objects on the stack per stack update. To take advantage of that, we implement the KRR stack as a simple array, where objects are ordered according to the stack order. When the object is referenced, we can find it in constant time using a hash table where a hash table entry holds a pointer to the array location. An object's stack distance is simply its

array index. On a stack update, first, we identify all swap positions, then we perform cyclic swapping on all marked positions. In our implementation, we adopt the spatial sampling technique described in Section 2.3.2. By default, we use a sampling rate of $R = 0.001$, but to ensure the accuracy of spatial sampling using SHARDS [7], a higher sampling rate of $R = 0.01$ is applied to workloads with relatively small working set sizes (less than 8M distinct objects).

For performance evaluation, we implement kRedis on top of Redis-4.0 with the default Jemalloc allocator. It uses KRR to model K-LRU policy with random sampling size K of 1, 5, 8, and 16, and chooses the best K on the fly for each tenant. It adopts the multi-dictionary scheme to accelerate tenant-level K-LRU random sampling and eviction process. Besides the original Redis as a primary baseline, we use pRedis [32] as a secondary baseline to evaluate the performance of kRedis in multi-tenant key-value cache. pRedis is based on the original Redis single-dictionary design and uses EAET to model exact LRU plus a bloom filter to discriminate key-value's tenant belonging. Both Redis and pRedis set K to 5 as default.

4.2 Experimental Evaluation

In order to evaluate the effectiveness of kRedis, we first give a brief description of the experimental setup and evaluation workloads. Second, we compare the performance

of Redis, pRedis, and kRedis. Next, we discuss the memory size impact, throughput, tail latency, and both time and space overhead of our design. Finally, we compare kRedis with DLRU and discuss the applicable fields of those two schemes.

4.2.1 Experiment Setup

We use two separate machines for our evaluation. Machine A is configured with an Intel(R) Xeon(R) Gold 5118 2.30GHz processor with 34 MB shared LLC and 188 GB of memory, and the operating system is Fedora 31 with Linux kernel 5.6.15. Machine B is configured with Intel(R) XEON(R) E5-2620 v4 2.10GHz processor with 20 MB shared LLC and 128 GB of memory, and the operating system is Ubuntu 18.04.6 LTS. All major evaluations are done on machine A, machine B is only used in Section 4.2.3.3.

A Redis cache server and multiple tenant front-ends are deployed on the local host. We implement Redis tenant front-end based on Hiredis library [71], which reads references from an evaluation trace and sends access requests to the Redis server on the fly. When the Redis server returns a miss, the tenant front-end will immediately follow a SET command to store the key-value pair into the server. With such a setup, the miss latency is simply the round-trip setback time between the tenant front-end and the Redis server. In most of our evaluations, we use various time delays to simulate

fetching items from the database, providing flexibility and variety in the miss latency setup to our evaluation. We also set up an evaluation environment with a real remote and local database to cross-validate the test case with the simulated environment. When evaluating multi-tenant scenarios, we set up multiple tenant front ends, with each tenant front end repeatedly sending requests from a workload until the server terminates.

4.2.2 Workloads

We use two different workloads for our evaluation:

† **MSR** MSR Cambridge suite [22] is a collection of block-level I/O traces from 36 volumes across 179 disks on 13 different enterprise data center servers in a Microsoft data center. We evaluate our model on all 13 traces, as well as the merged "master" MSR workload which is also used in Waldspurger *et al* [7]. The workloads encompass various applications such as home directories, project directories, hardware monitoring, firewall/web proxy, source control, web staging, media services, and more. The 13 workloads' reference counts range from 1 to 181 million, working set sizes range from 1 to over 1000 GB. More detailed trace information can be found in [72].

† **Twitter** Twitter cache traces [26] is a collection of one-week-long cache request

traces from 54 Twitter's in-memory caching clusters. We use sub-traces from Twitter clusters to evaluate our K-LRU model and kRedis performance. Each sub-trace consists of 100 million requests, detailed information of each workload including working set size, object size distribution, compulsory miss ratio, etc. can be found in [26, 73].

4.2.3 Access Latency

The memory allocation objective in Equation 4.3 is to minimize the overall miss latency or response time, so we use the average of tenants' mean access latency as the evaluation metric. The access latency is the wall clock time used by each access. We use the MSR and Twitter workloads in this evaluation. Redis cache maximum memory is set to 50% of the total working set size.

The workload sensitivity to the changes in sampling size K is the key to the exploration of the K-LRU miss ratio gap, and it is also the source of potential performance improvement from pRedis to kRedis. In order to compare the performance between the two, we first choose the workloads that are sensitive to the change of K and conduct case studies on a 4-tenant system. The MRCs of some example workloads are shown in Figure 4.2. We then stress the system by increasing the number of tenants to 15 using randomly selected traces from the Twitter suite.

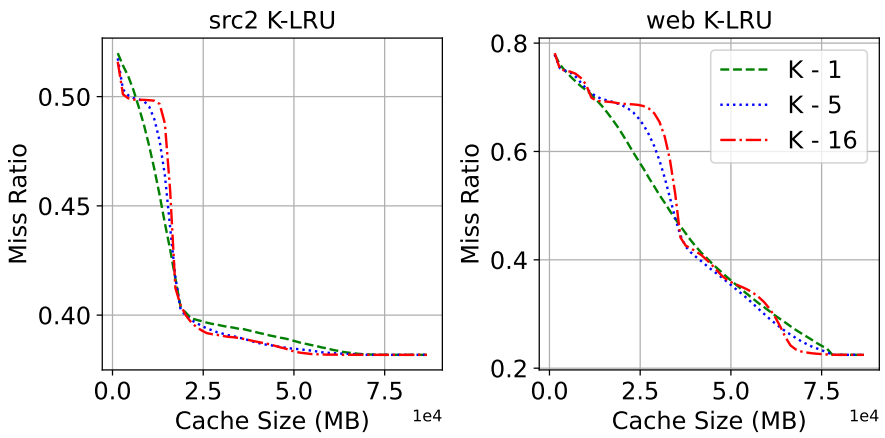


Figure 4.2: Workloads that are sensitive to the change of K. Miss ratio show gaps between different Ks at the same cache sizes.

4.2.3.1 4-Tenant Case Study

In this case study, we set up four tenants with MSR web and src2 workloads, representing fetching objects from web local-DB, web remote-DB, src2 local-DB, and src2 remote-DB, respectively. According to the access latency analysis of real Redis traces [32], the remote DB miss latency is typically distributed around 2000 μs. Therefore the miss latency is configured to 200 μs (local-DB) and 2000 μs (remote-DB), respectively. As shown in Figure 4.3, when compared to Redis, pRedis reduces the mean access latency of both web remote and src2 remote at the cost of a slight increase of the latency with the local-DB ones. And compared to pRedis, kRedis has further reduced the latency of web remote. Overall, the average access latency improvement of kRedis is 17.3% and 14.3% compared to Redis and pRedis, respectively.

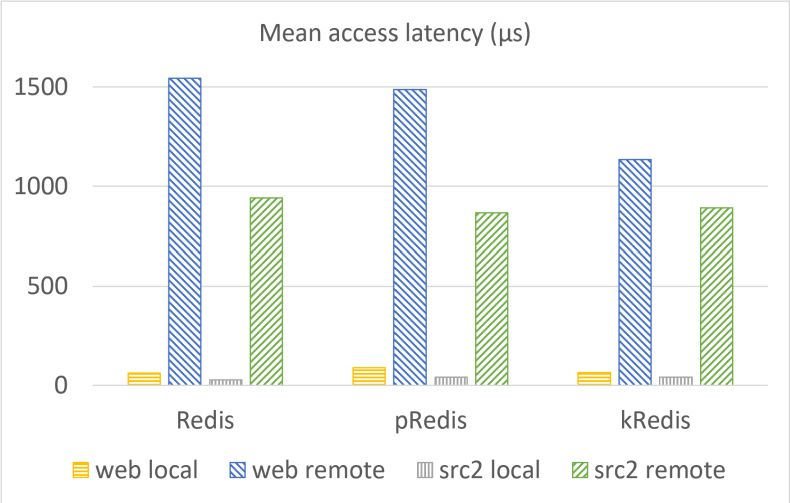


Figure 4.3: Average access latency reduction with 4 tenants loading MSR workloads.

Furthermore, in order to analyze the contribution of each optimization within kRedis, including the multi-dictionary design, the KRR model, and dynamic K configuration based on merged MRC, we use a series of variants to decompose their impact on performance, results are shown in Figure 4.4. First, pRedis, which considers miss latency, shows a 3.6% improvement over Redis, contributed by locality- and latency-aware memory partitioning. Second, we transform pRedis to the multi-dictionary design, which gains an additional 1.5%. Third, we use varKRR instead of EAET to model MRC under a fixed K of 5, which is the default setting in Redis and pRedis. Note that pRedis uses EAET to model exact LRU rather than K-LRU with $K = 5$, which could reduce MRC accuracy. With the more accurate KRR model, this variant of pRedis shows a 10.8% of improvement over Redis. Lastly, kRedis, combining optimization of KRR, dynamic K configuration, and multi-dictionary design, yields a 17.3% improvement over Redis. When comparing kRedis to the pRedis variant

with varKRR and fixed K of 5, the dynamic K configuration contributes a 6.5% improvement.

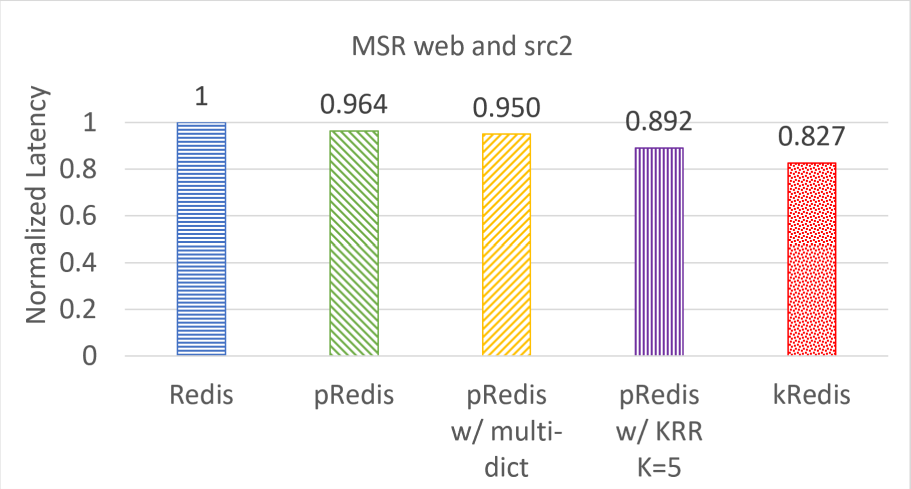


Figure 4.4: Impacts of kRedis optimizations on latency for MSR workloads, compared to Redis baseline.

In Figure 4.3, we observe that compared to pRedis, kRedis has decreased the access latency of web remote. To dig deeper into the process of dynamic K selection based on merged MRC, we provide snapshots of web remote K-LRU MRCs constructed by KRR in two individual intervals. Figure 4.5 (A) shows that in the 11th interval, the K-LRU MRC with $K = 1$ yields a lower miss ratio than other Ks at the partitioned memory of 800 MB, and kRedis sets K to 1 for the tenant. Figure 4.5 (B) shows that in the 15th interval, the K-LRU MRC with $K = 16$ shows a lower miss ratio than other Ks at the partitioned memory of 800 MB, and kRedis sets K to 16 for the tenant.

In another test case, we use Twitter sub-traces of cluster4.0, cluster29.0, cluster34.0,

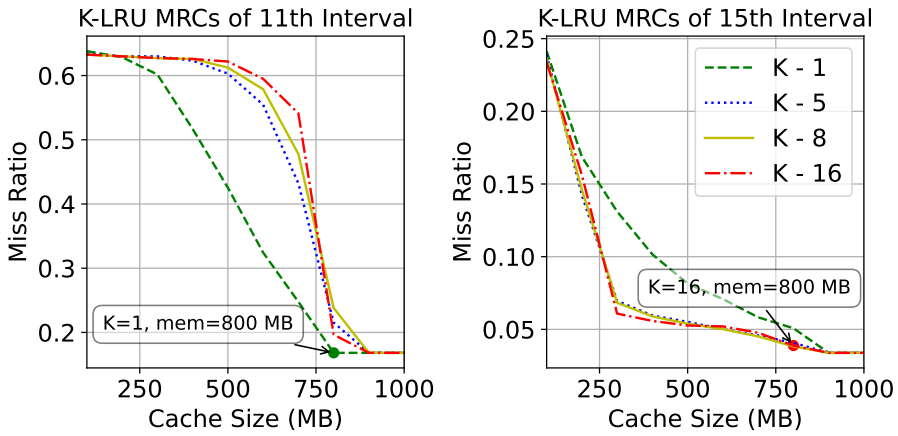


Figure 4.5: K-LRU MRCs of web remote in two evaluation intervals and K configurations. The periodical evaluation interval size is 1 million requests.

and cluster54.0 to generate references of 4 tenants, with their miss latency configured to 200 μs, 400 μs, 1000 μs, and 2000 μs, respectively, bringing more variety to the simulated miss latency. The average access latency improvement of kRedis is 27.0% and 16.7% compared to Redis and pRedis, respectively. Figure 4.6 summarises the latency results of pRedis, pRedis variants, and kRedis, compared to Redis baseline. First, we observe a 12.4% improvement related to locality- and latency-aware memory allocation. Second, with the multi-dictionary design, the improvement increases to 14.5%, Then equipped with varKRR for K-LRU MRC construction of K fixed to 5, it shows an 18.2% decrease in latency. Finally, kRedis which employs all optimizations shows a 27.0% improvement against Redis.

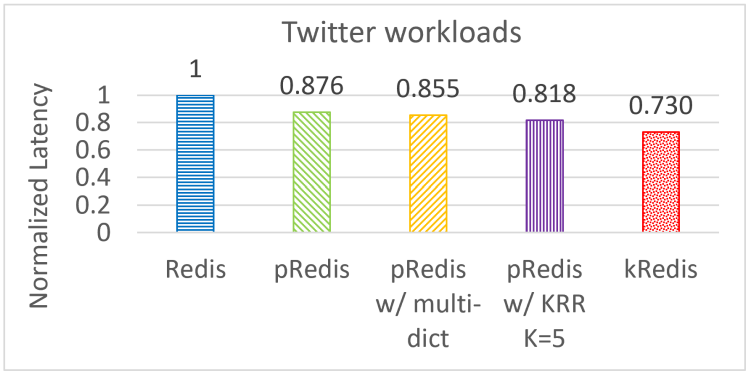


Figure 4.6: Impacts of kRedis optimizations on latency for Twitter workloads, compared to Redis baseline.

4.2.3.2 15-Tenant Case Study

To evaluate the performance of kRedis for a large number of tenants, we increase the number to 15, adding pressure to MRC construction, the cache partitioning algorithm, and the sampling-based eviction process. Now we assume each Twitter cluster trace comes from multiple tenants. Each reference of Twitter clusters includes a parameter named Client-ID which is the anonymous front-end service client who sends the request. We use this ID modulo 15 to generate a tenant ID. We randomly select 6 traces from Twitter: cluster4.0, cluster17.0, cluster18.0, cluster29.0, cluster44.0, and cluster52.0. For each workload, we set up the miss latency of each tenant based on the observed exponential distribution of miss latency of real-world Redis traces [32]: the miss latency of each tenant is set as 1 μs, 2 μs, 4 μs, 8 μs, 16 μs, ..., 4,096 μs, 8,192 μs and 16,384 μs, respectively. Figure 4.7 shows the average latency reduction of pRedis and kRedis, compared with the Redis baseline. kRedis reduces access latency up to

50.2% against Redis, and improves up to 24.8% when compared to pRedis.

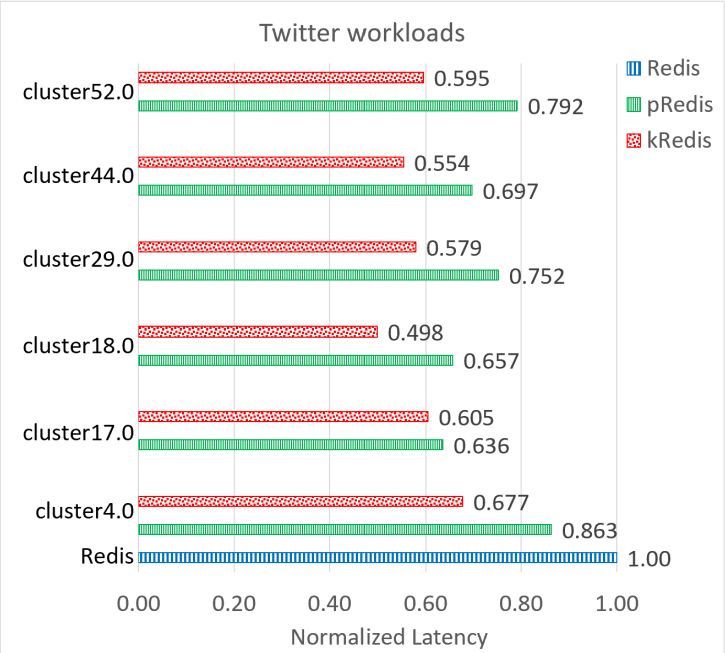


Figure 4.7: Average access latency reduction in Twitter workloads compared to Redis baseline. Tenant accesses are generated by partial references from a workload distinguished by Client-ID.

4.2.3.3 Real Back-End Database Case Study

In this section, we use two separate machines described in Section 4.2.1 running real back-end MySQL databases to cross-validate the test case shown in Section 4.2.3.1 with MSR workloads. Both Redis cache server and tenant front-end are running on Machine A. The web local-DB and src2 local-DB are also running on Machine A, and the web remote-DB, src2 remote-DB are running on Machine B. Both local and remote DB are deployed on MySQL Community Server 8.0.33. In this real MySQL

back-end DB setup, the miss latency of the local DB is distributed around 200 μs, and that of the remote DB is distributed around 2000 μs. Compared to Redis and pRedis, the average access latency improvement of kRedis is 17.2% and 13.3%, respectively. The results show no notable difference between the real back-end database and the simulated back-end database.

4.2.4 Impact of Memory Size

The max-memory setting of Redis impacts cache performance. Over-provisioned memory to a cache can lower the miss ratio of cache but at a higher cost on DRAM. In contrast, over-tight memory provision brings harm to cache performance. Intuitively, a tighter memory size introduces a higher miss ratio for all tenants, where kRedis has the potential to dynamically partition memory to meet the space requirement of tenants that are performance-critical for the optimization target. We evaluate kRedis performance on different max-memory settings to observe this trend based on the 4-Tenant test case of Twitter workloads in Section 4.2.3.1. The result shows that, compared to Redis, kRedis improves average access latency by 21.2%, 27.0%, and 49.5% with max-memory as 75%, 50%, and 25% of the working set size, respectively. The tighter limit on Redis memory brings higher performance improvement of kRedis against Redis.

4.2.5 Throughput

As described in Section 4.1.2, kRedis could achieve different optimization targets. We adopt Equation 4.4 and use the rate of reference hits against system time as the metric to evaluate throughput. We demonstrate the effect of kRedis with a 4-tenant case study. The four tenants load MSR workload mds, src2, stg, and web respectively. Cache max-memory is set to 50% of the working set size. All tenants' miss latency is set to 2000 microseconds simulating fetching objects from remote DBs. Table 4.1 shows kRedis improves the average throughput by 262.8% and 61.8% compared to Redis and pRedis, respectively. The similar setup for the 4-tenant case using 2 web and 2 src2 workloads shows similar results.

Table 4.1
Throughput (hits/sec) in MSR workloads

	mds	src2	stg	web	avg
Redis	1113	1455	2499	1701	1692
pRedis	1474	9839	1949	1920	3795
kRedis	485	22071	1485	515	6139

4.2.6 Tail Latency

kRedis has been proven to improve tenants' average latency as well as hit throughput, but we still need to figure out if the statistics tracking and memory allocation

of kRedis affects references' latency disproportionally. We evaluate the tail latency of kRedis in the first 4-tenant case study with MSR workloads discussed in Section 4.2.3.1. Table 4.2 shows the results. When comparing kRedis with Redis, there are no significant differences between the two for tenants with remote DB. However, since kRedis allocates more memory to the remote ones at the cost of increasing the miss rate of local ones, kRedis shows higher latency than Redis for the response times of src2-local.

Table 4.2
Request tail latency (μs)

		90th	95th	99th	99.9th
web-remote	Redis	2044	2049	2066	2457
	kRedis	2046	2056	2100	2270
src2-remote	Redis	2040	2047	2062	2129
	kRedis	2044	2054	2091	2219
web-local	Redis	223	228	241	267
	kRedis	224	231	253	307
src2-local	Redis	32	38	51	255
	kRedis	44	203	247	284

4.2.7 Time and Space Cost

In this section, we evaluate the time and space overhead of our approach. There are two sources of time cost: K-LRU MRC modeling with spatial sampling, and hash table resizing with multi-dictionary design in kRedis. The space cost also comes from two aspects: the implementation of the KRR stack and the multiple dictionaries.

First, we discuss time and space costs related to the KRR model. Then we analyze the overhead of multi-dictionary design.

To measure the efficiency of our KRR model and stack update mechanisms, we compare backward stack update methods (with/without spatial sampling) with the naive linear stack update method and the simulation/interpolation approach. We simulate K-LRU under 25 different cache sizes evenly distributed across its working set size. For demonstration purpose, we use the first one million references from MSR src1 trace, and set $K = 5$. Table 4.3 is a summary of the results. We see that the backward stack update method shows an 8247 times improvement over the linear stack update approach. When spatial sampling with $R = 0.01$ is applied, the running time is further improved by two more magnitudes. We also observe similar time cost improvement on other workloads.

Next, we use the merged "master" MSR trace to compare the running time of KRR+Spatial sampling with the existing LRU MRC approximation technique, SHARDS. Table 4.4 contains the running time for backward stack update KRR and SHARDS. The running time of KRR shown in Table 4.4 is the average across different Ks $(1, 2, 4, 8, 16, 32)$. The average running time for KRR with backward stack update and SHARDS is very close to the master trace in our test.

With the previous 15-tenant case study in Section 4.2.3.2, for all the 6 evaluated workloads, the total KRR time overhead including reference tracking and K-LRU

Table 4.3
Running Time Comparison for Processing One Million MSR src1 Requests

Stack Update Efficiency	
Methods	**Time (Sec)**
Simulation	26
Basic Stack	53606
Backward Stack Update	6.5
Backward+Spatial	0.07

Table 4.4
Master Trace Comparison

Merged-MSR Trace, Spatial Sampling Rate = 0.001		
Method	Backward+Spatial	SHARDS
Times (sec)	22.4	19.7

MRC modeling is in the range from 0.57% to 0.66% of total execution time.

The KRR stack is implemented as a simple array with a hash table where an entry of the hash table holds a pointer to an object location in the array. Then the total space overhead of the KRR stack is proportional to the total number of objects stored on the KRR stack. In our implementation, each object consumes 68 bytes including the hash table and other auxiliary entries. For variable object size-aware KRR, a 4 bytes field is needed to store the size of each object, the additional *sizeArray* consumes negligible space in comparison to the stack. After incorporating spatial sampling, the overall space overhead is further reduced by sampling rate R. Thus the estimated percentage of space overhead is *72 bytes * R / average object size*. For instance,

assuming $R = 0.001$, and the average size of objects is 200 bytes[1], then the space overhead is just 0.036% of the working set size.

In the following, we discuss the time and space overhead of multi-dictionary design in kRedis. The Redis hash table is capable of resizing itself according to the load factor. In the process of expansion or contraction of a hash table, Redis performs rehash operations which bring extra time overhead. In the original Redis, once the maximum memory is reached, the size of the single hash table is generally stable. But our multi-dictionary design may bring extra overhead while the size of each tenant's key space is changing according to the dynamic memory allocations. To measure the efficiency of our multi-dictionary design, we profile the time overhead of hash table resizing and compare it with Redis. We use the Twitter cluster 54.0 workload and emulate a 16-tenant system as described in Section 4.2.3.2. Redis' total rehashing takes 1 second out of 81199 seconds of running time, while kRedis' total rehashing takes 9 seconds out of 44474 seconds of running time. The rehashing time cost of multi-dictionary design is almost negligible as it accounts for only 0.02% of total running time.

In Redis, all key-values are stored as dict-Entry in the hash table, which is organized in a dictionary header structure containing type, pointer to hash table, rehash index, etc. In kRedis' multi-dictionary design, all space costs of actual key-value pairs are

[1]Many real in-memory cache workloads have much higher average key-value size [26]

the same as Redis, the only overhead is the metadata of the dictionary header, which is in a total of 176 bytes per tenant. Using the same instance in evaluating the space overhead of the KRR stack, assuming there are 15 tenants, then the space overhead for processing workload with 100 million distinct objects is just 1.32e-5% of the working set size.

4.2.8 kRedis vs DLRU

As described in Chapter 3, DLRU is also capable of reducing the overall access latency of a single-tenant, fixed memory size key-value cache by using miniature cache [67] to simulate the behavior of various K-LRU caches and explore the potential miss ratio gap of various sampling size Ks. kRedis can be downgraded to handle a single tenant, where KRR is applied to identify an optimal K. In this section, we first use a single-tenant environment to compare kRedis and DLRU. Then we extend DLRU to construct tenant MRCs in a multi-tenant key-value cache and use those to guide tenant memory partitioning. We use a multi-tenant test case to discuss the limitations of DLRU versus kRedis.

4.2.8.1 Single-Tenant Case Study

In this case study, we set up a fix-memory size Redis instance running a single tenant loading MSR workload. We use KRR and DLRU to guide the selection of K, respectively. We conduct two tests, using MSR workload src1 and web, respectively. We choose the Redis memory size as 30% of the workload's working set size where the miss ratios of various random sampling Ks show a large gap. The tenant's miss latency is configured to 2000 μs to represent fetching objects from the remote database. As shown in Figure 4.8, there is no notable difference in performance between DLRU and kRedis in the single-tenant use case. This indeed verifies the accuracy of the KRR model against simulation.

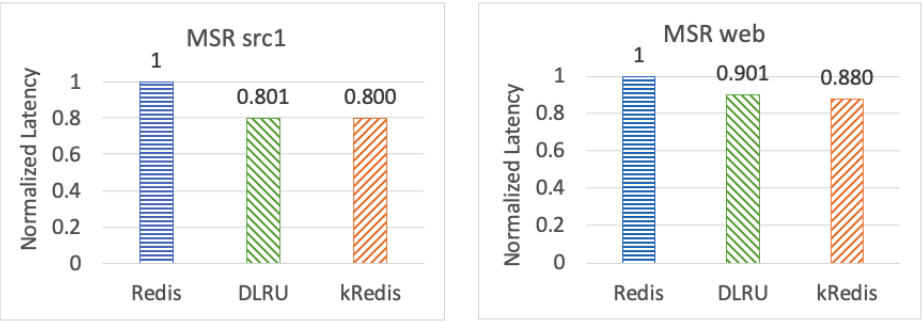


Figure 4.8: Average access latency reduction of DLRU and kRedis with single tenant loading MSR workload. There is no notable difference between the two schemes.

4.2.8.2 Multi-Tenant Case Study

To adopt DLRU for multi-tenant partitioning, we extend DLRU by generating MRCs through miniature cache simulation and interpolation. For each tenant, a selection of Ks, and a selection of cache sizes, we use D-LRU miniature cache to find the miss ratios. We then construct an MRC through interpolation for each K of each tenant, and use these MRCs to guide partitioning instead of the KRR MRCs. It is worth noting that interpolation may not be able to capture every inflection point on the MRC, thus losing accuracy in the constructed MRC.

To compare the extended DLRU against kRedis, we employ the same 15 tenants set up as the one used in Section 4.2.3.2 with Twitter cluster18.0 workload. In this test case, for each of the 15 tenants, we set up 20 cache sizes uniformly distributed across the range of Redis max-memory and provide 4 K options. Therefore, there are a total of $15 * 20 * 4 = 1200$ independent miniature caches in extended DLRU. Compared to Redis, extended DLRU and kRedis reduce the mean access latency by 45.2% and 50.2%, respectively. The time overhead of extended DLRU and kRedis are similar, which are 0.73% and 0.66% of total execution time, respectively, but extended DLRU shows higher space overhead than kRedis, which is 38 times that of kRedis.

It is worth mentioning that in extended DLRU the space and time overhead are directly associated with the number of cache sizes simulated. In order to obtain more

accurate K-LRU MRCs, more cache sizes need to be simulated for each tenant. And in a system with a large number of tenants and a greater cache max-memory size, the DLRU simulation overhead for the multi-tenant cache using K-LRU can only be higher. Thus, even though both DLRU and kRedis are efficient for the single-tenant use case, for multi-tenant memory allocation usage, an efficient one-pass MRC modeling algorithm such as EAET and KRR is preferred over interpolation.

4.3 Chapter Summary

In this Chapter, we present the design and implementation of kRedis, a lightweight memory partitioning scheme in multi-tenant key-value cache. Besides the capability of capturing trace locality and awareness of reference miss latency, it efficiently explores the potential miss ratio gaps of various sampling sizes in K-LRU. The evaluations over a variety of workloads show that our multi-tenant memory allocation approach achieves better performance than Redis and pRedis. The average access latency is reduced up to 50.2% and 24.8% when compared to Redis and pRedis, respectively. By adjusting the memory partitioning strategy, we show that the throughput is increased by 262.8% and 61.8% compared to Redis and pRedis, respectively. We also compare kRedis with extended DLRU and discuss the suggested application scenarios for the two.

Chapter 5

Tiered Memory Management for Multi-Tenant K-V Store

The increasing demand for memory as the k-v store scales to accommodate larger workloads, along with rising DRAM costs and challenges in technology scaling, has made memory a significant infrastructure expense in hyper-scale data centers, thus optimizing memory utilization becomes crucial for sustaining performance.

To address this challenge, a shift towards multi-tier in-memory k-v stores has risen. By adopting memory disaggregation architectures, particularly through technologies

like Compute Express Link (CXL), fast memory such as local DRAM, and high-density alternatives like NVM and shared memory from other devices can be integrated to construct tiered memory systems. These architectures separate memory resources into independent pools, interconnected by high-speed protocols like PCI Express (PCIe), allowing for increased memory utilization and reduced infrastructure costs through dynamic memory pooling. CXL facilitates a wide array of functionalities ranging from memory expansion within a single server to cross-node dynamic memory pooling, presenting a flexible and scalable solution to the rising costs and demand for memory.

However, the integration of shared slow memories introduces complexities due to their inherently lower bandwidth and higher latency compared to traditional CPU-attached DRAM. From an operating system's perspective, these are seen as CPU-free NUMA nodes with distinct memory characteristics. This architectural diversity necessitates efficient memory management strategies that can dynamically classify and migrate data between fast and slow memory tiers, optimizing for overall system performance. Despite the potential benefits, current page-level memory management approaches like HeMem [48], Nimble [46], and TPP [63], do not integrate application-level workload knowledge, facing challenges in efficiently managing the multi-tier memory. The need for application-aware memory management is becoming critical, especially in hyper-scale environments where application demands and access patterns rapidly evolve.

This chapter proposes sdTMM, a software-defined tiered memory management scheme for in-memory k-v stores in the multi-tenant environment.

Production in-memory caching workloads typically follow an approximate Zipfian popularity distribution, often exhibiting very high skew [26, 27, 74]. This means that hot items are concentrated in a small set. Consequently, similar to Google TMTS [65], the primary goal of sdTMM is to replace a portion of the conventional DRAM primary memory with CXL-shared memory while maintaining performance comparable to that of an all-DRAM system.

For simplicity, the DRAM of the host where the cache instance resides is called fast memory / fast tier, and all other memory shared by memory disaggregation techniques such as CXL interface is called slow memory / slow tier.

We implement a prototype of the sdTMM architecture using Cachelib [75], a pluggable caching engine. Our evaluations on an emulation hardware platform show promising results, with 80% of the DRAM memory being substituted with CXL-shared slow memory, the performance of sdTMM closely approximates that of an all-DRAM system. This demonstrates the effectiveness of the multi-tier memory management approach, where the integration of slower but greater capacity CXL memory with faster DRAM can maintain high-performance levels. Such findings underscore the potential of sdTMM to provide a viable alternative to traditional all-DRAM configurations.

5.1 sdTMM System Design

sdTMM is a software-defined tiered memory management architecture designed to seamlessly integrate fast local DRAM with slower shared memory. Its memory layers are transparent to applications and tenants, enabling key-value store applications to utilize the system without any adaptations. Unlike traditional hardware-based approaches, sdTMM manages memory at the application item level, providing enhanced visibility into application memory usage and access pattern changes. This allows for the implementation of sophisticated policies, including user-tuned item admission and eviction strategies, as well as fine-grained multi-tenant memory isolation and allocations. This architecture not only optimizes performance but also increases the flexibility and efficiency of memory management across diverse application demands.

Figure 5.1 presents a high-level overview of the sdTMM architecture, which is composed of four main components: memory hypervisor, CXL memory-sharing switch, fast DRAM memory tier, and slow memory tier as facilitated by the CXL memory-sharing switch. The total memory capacity available for the key-value store instance is the sum of the capacities from both the fast and slow tier. The fundamental strategy of sdTMM involves maintaining hot application-level data in the fast memory tier for optimal performance while relegating less frequently accessed data to the slow tier. sdTMM continuously monitors the dynamics of access patterns, allowing for the

migration of data between tiers based on changes in data hotness. While evictions are predominantly managed within the slow tier, spikes in new item influxes can also trigger evictions from the fast memory tier. Memory allocation in multi-tenant environments is restricted to the fast tier, where the local DRAM size is more limited and provides lower latency and higher bandwidth compared to the slow memory tier, essential for maintaining superior system performance.

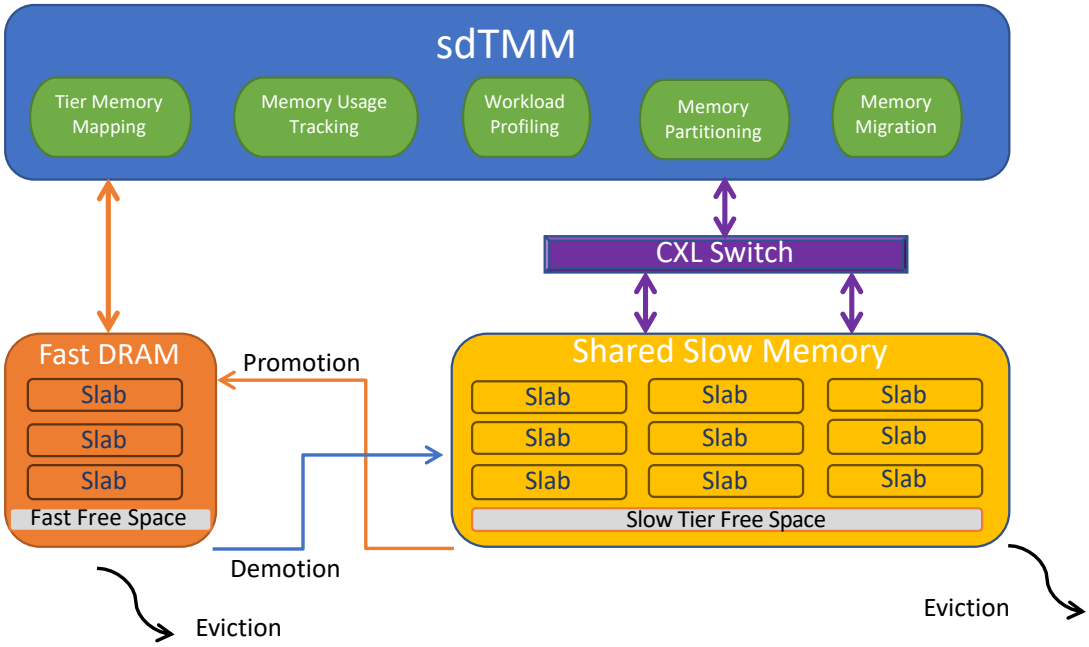


Figure 5.1: sdTMM Architecture.

The sdTMM hypervisor consists of five modules performing key functions essential for optimizing memory management across dual tiers.

The first module, tier memory mapping, involves managing the placement of application-level data across the memory tiers. In both fast and slow memory tiers,

application key-value items are directly addressed. This is achieved through a universal hash table which contains compressed pointers pointing to valid memory allocations, ensuring that reference access is correctly mapped to the appropriate memory tier and allocations.

The second module, memory usage tracking, monitors memory usage and collects statistics of applications for performance analysis, it also confines each tenant's memory usage within the partitioned configurations in the multi-tenant environment. This allows for a more informed assessment of memory distribution and utilization.

The third and fourth modules are workload profiling and memory partitioning. Ideally, in real-time, the hypervisor utilizes spatial sampling to profile tenant workloads, and constructs MRCs for each tenant on the fly using efficient MRC modeling techniques to guide the partitioning of the fast memory tier. This module is also designed to accommodate read-in fast tier allocation configurations guided by MRCs generated by other profilers or constructed offline, enhancing flexibility in memory management strategies. Details is discussed in Section 5.1.5

The final module in the sdTMM framework is application-level data access tracking and migration management. Within this context, demotion refers to migrating warm data from the fast memory tier to the slow memory tier, and promotion involves moving newly identified hot items in the opposite direction. To facilitate these processes, sdTMM employs three types of background workers dedicated to managing

data promotion, demotion, and proactive eviction. The hypervisor actively monitors changes in application access patterns, continuously identifying data that may require promotion or demotion. These workers operate in the background, asynchronously moving data in batches between the memory tiers as needed. This ensures that the system can adapt dynamically to varying data access patterns without significant delays or disruptions in performance. The system configuration can be adjusted based on application characteristics to optimize overall system performance, enhancing efficient data handling and system responsiveness. Further details on these processes are outlined in Section 5.1.3 and Section 5.1.4. This adaptive approach ensures that sdTMM not only meets the current data access demands but also remains efficient and responsive under changing workload conditions.

The architecture of the sdTMM memory tiers is designed to optimize data management and access, structured as follows:

The fast memory tier consists of the following parts:

1. Main Memory Space: This segment stores tenant objects, providing robust and rapid access services.

2. Free Space: Serving as a logical buffer, this area is reserved for quickly absorbing new incoming data items set by tenants, and those newly identified hot data migrated from the slow memory tier.

The slow memory tier consists of the following parts:

1. Main Memory Space: Similar to the fast tier, this segment stores tenant objects but serves at a slower access speed.
2. Free Space: Acts as a logical buffer to accommodate batches of warm data migrated from the fast memory, aiding in efficient tier data transitions.

Together, these components form a cohesive system that dynamically manages data placement and migration across the memory tiers, balancing performance with efficient resource utilization.

5.1.1 Item Admission and Eviction

The Least Recently Used (LRU) replacement policy has demonstrated robust performance in capturing workload locality. Numerous modern modeling techniques are designed specifically for LRU, enabling the efficient construction of its MRC [2, 9, 13, 14, 15, 24, 32]. Consequently, sdTMM adopts LRU as its principal eviction algorithm. Within the memory tiers, items are approximately ordered by their last access time, with further discussion on this mechanism in Section 5.1.2

In the sdTMM framework, new items generated by applications, as well as those reinserted following cache misses in the look-aside cache mode, are directly admitted

into the fast memory tier. Once the capacity of the fast memory tier is reached, evictions are initiated within this tier to accommodate incoming new items, and the least recently accessed items are evicted. Similarly, when the slow memory tier reaches its capacity limit, evictions are triggered to make room for the items being migrated from the fast tier. We name this process as reactive evictions. However, due to the strategy of proactive memory collection for the duo memory tiers described in 5.1.3, such reactive evictions are generally infrequent.

Drawing inspiration from the filter queue design in FIFO [74] and aiming to enhance the efficiency of evictions, our reactive eviction strategy in the fast memory tier involves directly removing the least recently accessed items from the entire k-v store, rather than migrating them to the slow memory tier. Studies of production workloads indicate a prevalence of "one-hit-wonder" items in real-world traces [73, 74]; excluding these items can boost cache efficiency. Thus, in sdTMM, the fast memory tier serves a dual purpose: it not only stores data but also acts as a filter cache.

Moreover, if items evicted from the fast tier were moved to an already full slow memory tier, it would necessitate further evictions within that tier to create space, a process we term "cascade evictions". Such cascade evictions complicate and delay the admission of new items into the system. By opting to remove items entirely during fast-tier reactive eviction, rather than relocating them within the memory hierarchy, we prevent the potential slowdown associated with these additional evictions,

maintaining a more streamlined and efficient admission and eviction process.

5.1.2 Insertion Point Design

To efficiently filter out one-hit wonders in the fast memory tier, we propose a modification to the standard LRU insertion policy. Instead of inserting new items at the head of the LRU list, we suggest positioning these items near the tail of the list. This strategy allows one-hit wonders to quickly move to the LRU tail and be expelled from memory sooner, while frequently accessed items, upon subsequent references, are moved to the head of the LRU list and remain unaffected by this change.

Conversely, when items are demoted from the fast to the slow memory tier, placing them at the head of the LRU list in the slow tier could inadvertently lead to their premature promotion back to the fast tier during promotion scans. This would result in wasteful consumption of the bandwidth between the two tiers—a phenomenon we term as "ping-pong effect" in item migration.

Drawing inspiration from the Insertion Point (IP) design of Cachelib engine [27], we adopt differentiated IP positions for the fast and slow memory tiers to tailor our approach to both expedite the filtering of one-hit wonders and mitigate the ping-pong effect. Specifically, in sdTMM, the insertion point in the fast memory tier is set at 1/4 size of the LRU list from the tail, and in the slow memory tier, it is positioned

at the midpoint of the LRU list.

5.1.3 Proactive Memory Collection

As presented in the architecture of sdTMM, both the fast and slow memory tiers include a free space, reserved to swiftly accommodate new data items for the respective tier. This free space is proactively maintained through the coordination of demotion and eviction mechanisms, ensuring that memory is efficiently managed and available when needed.

Drawing inspiration from Intel's implementation of the multi-tier Cachelib engine [75], sdTMM utilizes two configurable thresholds to manage the free space within each memory tier: the lowMem and highMem thresholds. In the fast memory tier, the background demotion worker periodically checks the current percentage of free memory. Should this percentage fall below the lowMem threshold, the worker is activated to proactively migrate warm items from the fast to the slow memory tier in batches. This demotion process continues until the percentage of free memory reaches the highMem threshold, at which point the demotion worker ceases operation. Details of the demotion mechanism are discussed in Section 5.1.4.3.

Similarly, in the slow memory tier, the background eviction worker monitors the current percentage of free memory. If this percentage drops below the lowMem threshold,

the worker is triggered to begin evicting cold items in batches to free up space. This eviction process continues until the memory percentage reaches the highMem threshold. Details about this process are presented in Section 5.1.4.4.

In the default configuration of sdTMM, the lowMem and highMem thresholds are set to 2.0% and 5.0% respectively.

In summary, the proactive memory collection process in sdTMM is characterized by a directional flow of items: from the fast tier to the slow tier, and from the slow tier to eviction from the cache. This method ensures that both the fast and slow memory tiers remain agile and prepared to quickly accept new data items that are destined to become residents of the respective tiers. This structured flow facilitates efficient memory management, enabling each tier to adapt swiftly to new data demands and maintain high performance.

5.1.4 Data Migration

The data migration module in sdTMM actively monitors the "hotness" of items currently residing within each tier, identifies new hot items, and adjusts their placement within the memory hierarchy to leverage the advantages of each tier, thereby enhancing overall system performance. Specifically, the fast memory tier offers lower access latency and high bandwidth but has a limited size, whereas the slow memory

tier presents the opposite characteristics—higher latency and lower bandwidth but greater capacity.

The core principle of data migration is to dynamically track changes in data item hotness as influenced by shifts in application access patterns. The goal is to place hot items in the fast memory tier and move others to the slow memory tier. This stratified management strategy ensures that the system efficiently utilizes the distinct properties of each tier to optimize performance.

This migration process is supported by two critical components: the first involves the identification of item hotness, and the second encompasses the execution of efficient data demotion and promotion actions. Together, these components facilitate a responsive and adaptive caching system that can swiftly adjust to changing data access demands.

5.1.4.1 Item Hotness Identification

In the sdTMM framework, items within both the fast and slow memory tiers are approximately ordered based on their last access time, adhering to the LRU eviction policy. Items that have been accessed recently are positioned near the head of the LRU list, while others gravitate toward the tail.

Intuitively, one can use an item's last access time as a direct measure of its "hotness" to determine promotion eligibility, specifically, by comparing the last access time of items at the head of the slow tier's LRU list with those at the tail of the fast tier's LRU list, which we named as naive TMM system (naiveTMM). However, this approach has limitations. For instance, a sequential scan access pattern can undermine this method. In such scenarios, each item is accessed only once, yet every subsequent item's access time appears more recent than the last, potentially leading to inappropriate promotions from the slow to the fast tier.

To address this challenge, we introduce reuse time as a supplementary metric for assessing item hotness. Reuse time is defined as the time difference between an item's current access time and previous access time.

In the fast memory tier, upon items' re-accesses, we employ a rolling window to maintain statistics on the reuse time, which serves as an index of the residents' hotness level. Currently, we use rolling Window Mean Reuse Time as the hotness threshold for making informed promotion decisions. Conversely, in the slow memory tier, each item's reuse time is recorded as metadata upon re-access. Importantly, given that re-accesses occur more frequently in the fast tier than in the slow tier, we do not update their reuse time in the metadata for fast-tier items. Instead, we only update the window statistics, which reduces the overhead associated with accesses in the fast memory tier. This method ensures a more accurate and efficient determination of

item hotness, facilitating better management of data migration between tiers.

5.1.4.2 Background Promotion

In sdTMM, we deploy periodic background promotion workers to scan for newly hot items in the slow memory tier and migrate them to the fast memory tier asynchronously in batches. These workers initiate their search from the head of the LRU list within the slow tier. They compare each item's reuse time against the hotness threshold established for the fast tier. Items exhibiting reuse times shorter than this threshold are marked and accumulated in a batch.

Once a pre-configured number of candidates has been identified and marked for promotion, the promotion worker coordinates with the memory allocator to secure the necessary memory space within the fast tier. The marked items are then moved from the slow tier to the fast tier in organized batches by invoking the standard Linux memmove library method. Thanks to the proactive memory collection mechanism, outlined in Section 5.1.3, there is typically a reserve of free space in the fast tier, ensuring that there is sufficient memory available for the newly promoted items. In instances where the available space in the fast tier is insufficient to accommodate all the promoted items, the worker will proactively demote certain items back to the slow tier to free up the necessary space before proceeding with the promotions.

5.1.4.3 Background Demotion

The design of the demotion process mirrors that of the promotion scheme, employing the periodic background demotion worker to facilitate the migration of items from the fast tier to the slow tier in batches. The demotion worker initiates their process by scanning from the tail of the fast tier's LRU list. Once a pre-determined batch number of items has been selected, the worker coordinates with the memory allocator to secure the required memory space in the slow tier. The marked items are then moved to the slow tier using the Linux standard memmove. Although it is uncommon for the slow tier to lack sufficient memory—due to the proactive memory collection scheme described earlier—there are instances where available space may be inadequate. In such cases, the demotion worker takes proactive measures by evicting a necessary number of items from the tail end of the slow tier's LRU list. This action ensures that there is enough space to accommodate the newly demoted items.

5.1.4.4 Background Eviction

We utilize periodic background eviction workers to monitor and maintain the available free space in the slow memory tier, as detailed in Section 5.1.3. The worker operates under the guidelines established by the lowMem and highMem thresholds. Similarly to the demotion process, the background eviction worker scans for candidates from

the tail of LRU list within the slow memory tier, which are marked and accumulated in a batch. Once a pre-configured number of items has been designated, the worker proceeds to evict them from the slow tier, effectively removing them from the entire k-v store.

5.1.5 Multi-Tenant Memory Partitioning

Similar to the scenario described in Section 4.1, sdTMM is capable of serving multiple applications with a single cache instance. This architecture allows for isolating each tenant's workload, particularly in terms of eviction and memory footprint, ensuring that memory resources are managed distinctly for each tenant. In sdTMM, the system enhances the overall cache's performance by maintaining separate eviction domains for each tenant. When an application's dedicated memory space reaches capacity, sdTMM restricts eviction to that specific tenant's domain, avoiding interference with the memory allocated to other tenants. This is particularly beneficial given that the MRC often varies across different workloads. Using memory partitioning to isolate these workloads ensures that the performance demands of one application do not detrimentally impact others.

Unlike kRedis discussed in Secion 4.1, which operates solely with a DRAM memory tier, sdTMM's memory architecture integrates both fast DRAM and CXL-shared slow

memory, yet remains transparent to applications, requiring no adjustments for their operation on sdTMM.

The process of memory partitioning in sdTMM is akin to that in kRedis presented in Section 4.1. It involves monitoring each tenant's workload using the spatial sampling techniques described in Section 2.3.2, and constructing an MRC for each application using the KRR algorithm outlined in Section 2.3.1. The partitioning algorithm, presented in Section 4.1.2, is then applied to determine the optimal memory distribution among all tenants. sdTMM supports flexible memory management strategies by allowing load allocation configurations guided by MRCs generated by other profilers or constructed offline.

Following the allocation decisions, the memory usage tracking module actively monitors each tenant's memory usage. This module coordinates with the memory allocator to ensure that each tenant's memory usage remains within the configured limits. Currently, sdTMM only partitions tenant memory in the fast memory tier, considering that the fast tier is size-constrained and performance-critical for the whole k-v store system compared to the slow tier.

Should there be a discrepancy between the current tenant memory distribution and the latest allocation decisions in the fast tier, sdTMM utilizes background memory resizing workers. These workers adjust each tenant's memory allocation asynchronously,

expanding or contracting the allocated memory based on the new partitioning configuration to seamlessly manage changes in tenant requirements and workload demands.

5.1.6 Implementation

The memory management for both fast and slow tiers utilizes a slab-based approach [76]. Within each tier, memory is divided into independent pools to provide isolation for different tenants' data.

Each pool consists of distinct allocation classes, where items are logically organized by size. This organization minimizes fragmentation and facilitates pointer compression through an index offset in the slab.

A slab is a physically contiguous memory unit of 4 MB. Each slab is assigned to one allocation class within a particular pool or remains unassigned. Allocation classes have multiple slabs, which do not need to be physically adjacent.

Memory allocations for application items are made from these slabs. All allocations within a slab are of a fixed size, determined by the allocation class, and belong to a single memory pool. The minimum allocation size within a slab is 64 bytes, and the sizes of allocation classes increase following a geometric sequence.

Items within an allocation class are organized using an LRU doubly-linked list. Upon eviction, the least recently used item from that allocation class is removed. Each tier, pool, and slab class maintains its independent LRU lists.

The memory mapping and caching indexes are implemented using a hash table. Application items in both fast and slow memory spaces are directly addressed through this hash table, which maps keys to compressed pointers that indicate valid memory allocations. These compressed pointers combine the tier index, slab index, and allocation index within the slab.

5.2 Experimental Evaluation

To evaluate the effectiveness of sdTMM, we first provide a brief description of the experimental setup, including hardware and software platform and configurations. Then we introduce evaluation workloads. Third, we evaluate the performance of sdTMM. Our primary objective of sdTMM is to replace a fraction of the conventional DRAM primary memory with CXL-shared slow memory and achieve similar performance to that of an all-DRAM system (allDRAM). Finally, we discuss the tail latency, time, and space overhead of our design.

5.2.1 Hardware Platform

Given the current unavailability of commercial CXL interface hardware, we have opted to emulate our multi-tiered memory architecture using a Non-Uniform Memory Access (NUMA) architecture for evaluation purposes. The server employed in our study is equipped with an Intel(R) Xeon(R) Gold 5118 processor, operating at 2.30GHz. It includes 2 sockets, encompassing 48 logical cores divided into 2 NUMA nodes, each comprising 24 logical cores. The server has 34 MB of shared Last Level Cache (LLC) and a total of 188 GB of memory, with each NUMA node having 94 GB of directly attached memory.

For the deploying of the sdTMM cache instance, we utilized the numactl [77] to bind the cache process specifically to NUMA node 0. This configuration leverages the memory directly attached to NUMA node 0 as the fast memory tier while treating the memory attached to the far-side NUMA node 1 as the slow memory tier. In this setup, for both small (128 Bytes) and large (1024 Bytes) object sizes, under random read and write conditions, the slow tier exhibited twice the latency of the fast tier. Additionally, the bandwidth of the fast tier was found to be 2.4 times greater than that of the slow tier. This arrangement effectively mirrors the intended behavior of a CXL-switched environment. Access to the NUMA memory is facilitated through the Linux standard NUMA memory library APIs, which are invoked to emulate the

behavior of the CXL switch. The server runs Fedora 38 with the Linux kernel version 6.3.7.

Should commercial CXL memory-sharing hardware become available, we are prepared to migrate and validate the sdTMM architecture in a genuine CXL-enabled environment,

5.2.2 Software Platform

To rapidly prototype sdTMM architecture, we selected Intel Cachelib [75], a forked variant of Meta Cachelib [76]. Meta Cachelib is a highly adaptable caching engine designed to construct and enhance high-performance cache services. It is a thread-safe, scalable, C++ library that underpins the fundamental caching mechanisms. Developers leverage Cachelib to explore and prototype new caching heuristics and frameworks efficiently [27, 74, 78, 79].

Additionally, it includes CacheBench, a benchmarking and stress-testing tool that evaluates cache designs against industry-standard cache workloads. In our evaluations, we use CacheBench to load workloads and stress our sdTMM system.

Since its initial deployment in 2017, CacheLib has become an integral component of

more than 70 services at Meta, including the content delivery network (CDN), social-graph cache, application look-aside cache, and block-storage systems. Intel's variant of Cachelib advances this framework by incorporating multi-level caching capabilities, such as L1 DRAM and L2 CXL shared memory. It also introduces data promotion and demotion APIs. However, its demotion process bears potential for cascade evictions, as discussed in Section 5.1.1, and its promotion currently employs naive promotion approach discussed in Section 5.1.4.1.

5.2.3 Workloads

We use distinct workloads for our evaluation:

† **YCSB** Yahoo Cloud Serving benchmark [80] is a widely recognized benchmark that offers a suite of six core workload types. For our analysis, we focus on Workload C and Workload E from this suite. Workload C is a read-only workload that adheres to a Zipfian distribution, which is commonly used to simulate access patterns in web applications where some items are far more popular than others. Workload E is a scan-dominant workload characterized by its initial Zipfian distribution to select the first key in a range, followed by a uniform distribution to determine the number of objects to scan. For Workload E, we configure the maximum scan length to equal the number of distinct objects in

the workload. We use workloads with four different skewness parameters (α values), specifically 0.5, 0.8, 0.99, and 1.2. And we use uniformed value size of 300 Bytes in all workloads.

† **MSR** MSR Cambridge suite [22] originates from 13 enterprise data center servers. The suite captures a variety of applications including home directories, project directories, hardware monitoring, and web services. The traces vary significantly, with reference counts ranging from 1 to 181 million and working set sizes from 1 GB to over 1000 GB. Workload details can be found in [72].

† **Twitter** Twitter cache traces [26] include a week-long collection of cache request traces from 54 of Twitter's in-memory caching clusters. We utilize sub-traces from these clusters, each consisting of 100 million requests. Detailed information including working set size, object size distribution, one-hit-wonder ratio, etc. can be found in [26, 73].

5.2.4 Throughput & Hit Rate

Production in-memory caches typically operate at low miss ratios to ensure high performance [26, 74]. Therefore, in our evaluation, we use the memory over-provisioned all-DRAM single-tier system (allDRAM) as the baseline, we also include the results of naiveTMM described in Section 5.1.4.1 as a secondary comparison for illustrative

purposes.

In sdTMM we assume that the fast tier space is limited, while the memory of the slow tier is over-provisioned. sdTMM is in look-aside cache mode so that on cache misses, the CacheBench will immediately setback the missing key-value pairs. We use get throughput and fast-tier hit rate as the metrics to evaluate system performance.

The evaluations are organized as follows:

1. To determine if sdTMM achieves its design objective of parity performance with an over-provisioned all-DRAM system, in Section 5.2.4.1 we use randomly selected workloads from real-world production traces, including the Twitter suite and MSR suite, for a comprehensive evaluation in a single-tenant system.

2. To assess the performance of sdTMM's fast-tier memory partitioning, in Section 5.2.4.2 we compare the sdTMM MRC-guided partition against free competition and equal partition strategies in a multi-tenant environment.

3. To evaluate sdTMM's effectiveness in dynamic hot item detection and placement in memory tiers, we conduct a case study in Section 5.2.4.3 using a synthetic workload with hot item pattern changes across different phases, investigating sdTMM's performance in each phase.

4. Finally, we evaluate sdTMM in more complex scenarios using workloads that feature phase-changing and multi-tenancy in Section 5.2.4.4.

5.2.4.1 Single-Tenant Case Study

In this section, we use randomly selected workloads from real-world production trace suits, Twitter and MSR, to evaluate sdTMM performance under a single-tenant use case. The fast-tier memory size is configured as 20% of the working set size. Figure 5.2 shows the results. The average throughput impact of sdTMM is 13.0%, with a minimum throughput degradation of 2.2% compared to allDRAM.

We observe that for Twitter cluster44.0, cluster19.0, and MSR src1, the degradation exceeds 20%. In the case of Twitter cluster44.0 the uniformity in k-v sizes across two sizes results in only two slab classes in sdTMM. Consequently, there are only two LRU lists in each memory tier, leading to high costs in the LRU operations, Figure 5.3 shows the CPU time spent on LRU update operations measured using the Intel VTune profiler [81]. For Twitter cluster19.0, the alpha value is 0.735, indicating that frequently accessed items are not concentrated in the key space. The fast-tier hit rate of sdTMM is 36.13%. Similarly, in the case of MSR src1, the alpha value is as low as 0.4, and the fast-tier hit rate is 29.21%, this low fast-tier hit rate has a large impact on throughput.

We also observe significant throughput impacts on naiveTMM for Twitter cluster52.0, cluster29.0, cluster19.0, cluster4.0, and MSR prxy. This is because, in naiveTMM, promotion is not regulated by the refined hotness threshold as applied in sdTMM.

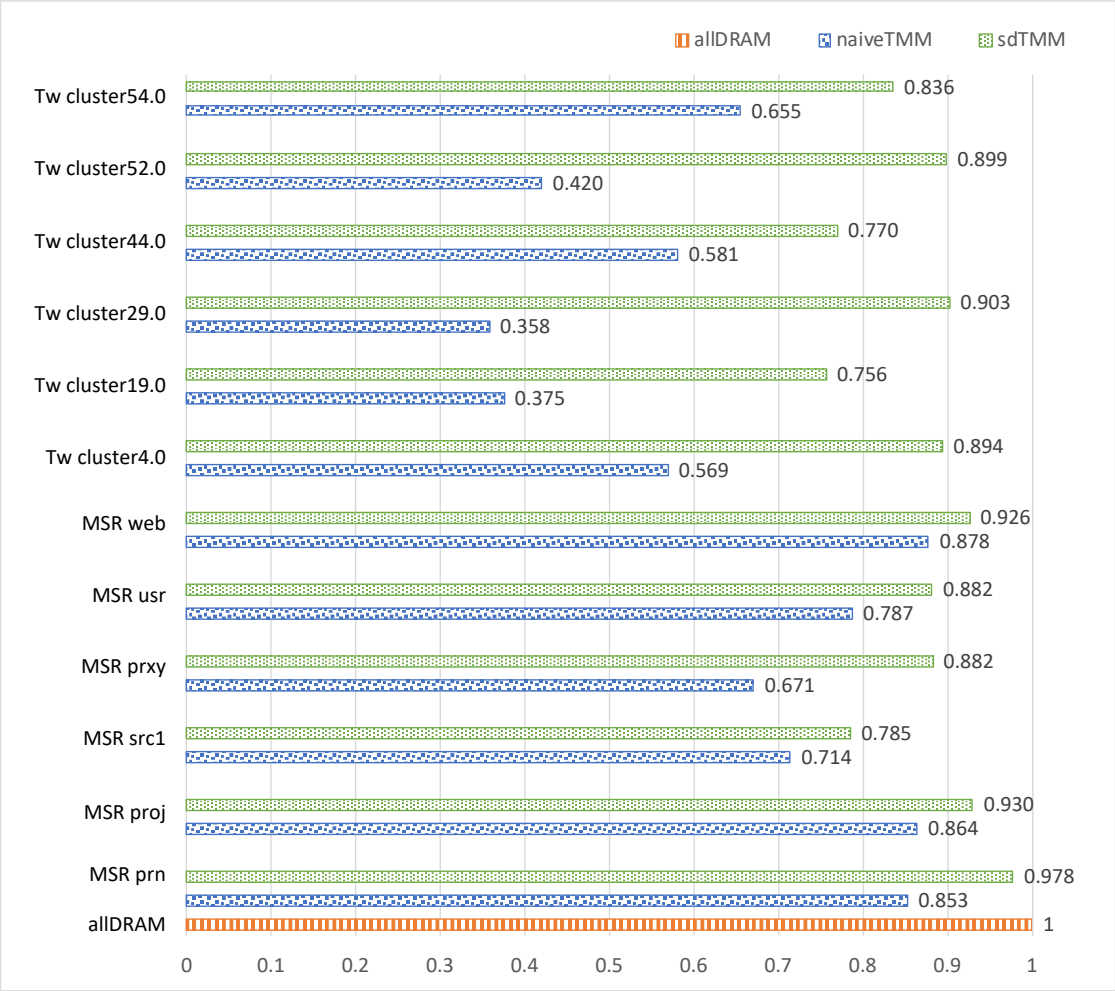


Figure 5.2: Throughput impact in Twitter and MSR workloads compared to allDRAM.

When the fast-tier free space is insufficient to accommodate the batched promotion items, this promotion triggers prerequisite demote operations from the fast-tier to the slow-tier to free up space. This chained item migration negatively affects system performance. Figures 5.4 illustrate an example of Twitter cluster29.0, showing the CPU time spent on item movement where demotion is triggered by the promotion process, as measured by the Intel VTune profiler. This highlights the importance of identifying the dynamic hotness level in the fast tier and throttling the item promotion

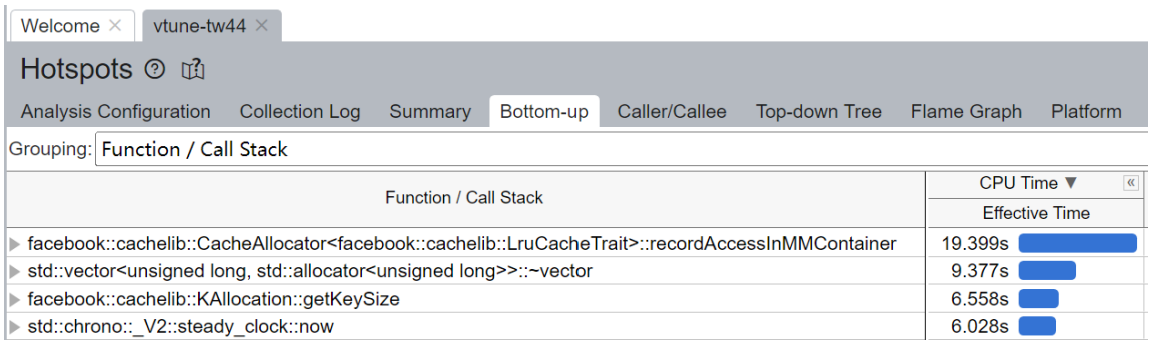


Figure 5.3: High CPU time spent on LRU updates with sdTMM in Twitter cluster 44.0.

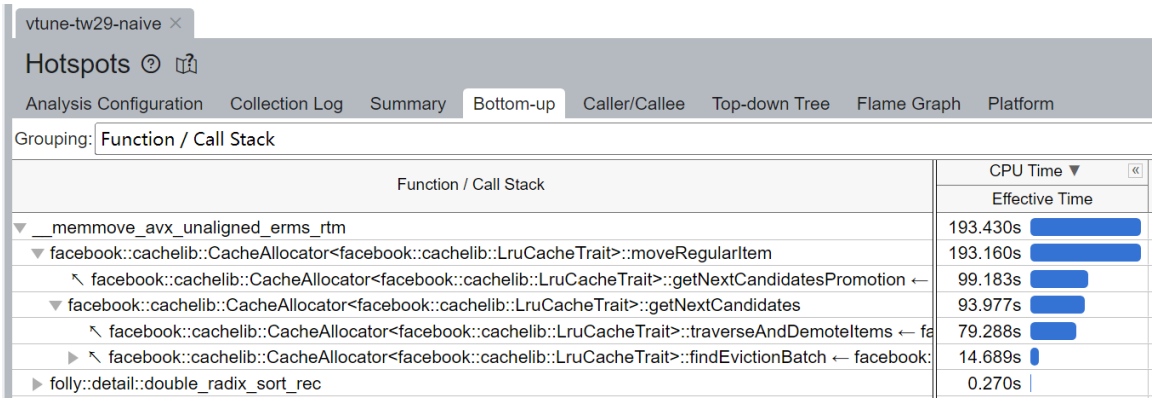


Figure 5.4: High CPU time spent on chained item movements of naiveTMM with Twitter cluster 29.0

among memory tiers.

5.2.4.2 Multiple-Tenant Case Study

To evaluate the performance of sdTMM's fast-tier memory partitioning, we compare three different strategies: MRC-guided partition, free competition, and equal partition. In the MRC-guided partition strategy, we follow the steps outlined in Section 5.1.5. For simplicity, tenant MRCs are constructed offline using the KRR

model [25], and the partitioning algorithm presented in Section 4.1.2 is applied to calculate memory allocations for each tenant, sdTMM then follows this configuration in the fast tier memory allocations, ensuring isolated memory spaces in the fast tier. In the equal-partition strategy, sdTMM allocates equal memory for each tenant, while in the free-competition strategy, tenants compete for fast memory space without constraints. The slow-tier space is over-provisioned to ensure that variations in the partition strategies impact only the fast-tier,

We first use two tenants, each running a YCSB workload. The Tenant 1 loads YCSB workload features an *alpha* value of 0.8, with 100 million references and 10 million records. The Tenant 2 loads YCSB workload features an *alpha* value of 1.2, with the same reference number and record count as Tenant 1. For the three partitioning strategies applied sdTMM, the fast-tier memory size is set to 50% of the working set size. Figure 5.5 shows the results. Compared to the allDRAM system, the throughput degradation of MRC-guided partition, free competition, and equal partition are 4.2%, 5.5%, and 8.0%, respectively. The fast-tier hit rate for allDRAM, MRC-guided partition, free competition, and equal partition are 92.92%, 80.22%, 78.40%, and 68.56%, respectively. MRC-guided partition achieves the highest fast-tier hit rate among the three strategies, resulting in the lowest performance impact.

Next, we increase the number of tenants to four and introduce a noisy neighbor running a scan pattern workload. In this scenario, the free competition memory partition

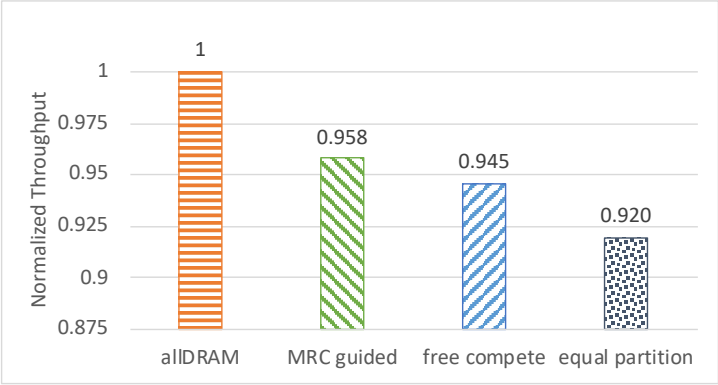


Figure 5.5: Normalized throughput of 2 tenants with fast-tier memory partitioning in YCSB workloads

strategy may cause issues, as the noisy neighbor will compete for fast-tier memory without contributing to the hit rate. The first tenant runs a scan workload, while the other three tenants run YCSB workloads with *alpha* values of 0.5, 0.99, and 1.2, respectively. The fast-tier memory size is set to 10% of the working set size. Figure 5.6 shows the results. Compared to the all-DRAM system, the throughput impact of the MRC-guided partition, free competition, and equal partition strategies are 2.3%, 5.0%, and 3.6%, respectively. The fast-tier hit rates for all-DRAM, MRC-guided partition, free competition, and equal partition are 69.55%, 61.48%, 55.76%, and 57.13%, respectively. The MRC-guided partition demonstrates the best performance, while free competition performs worse than equal partition due to the presence of the noisy neighbor.

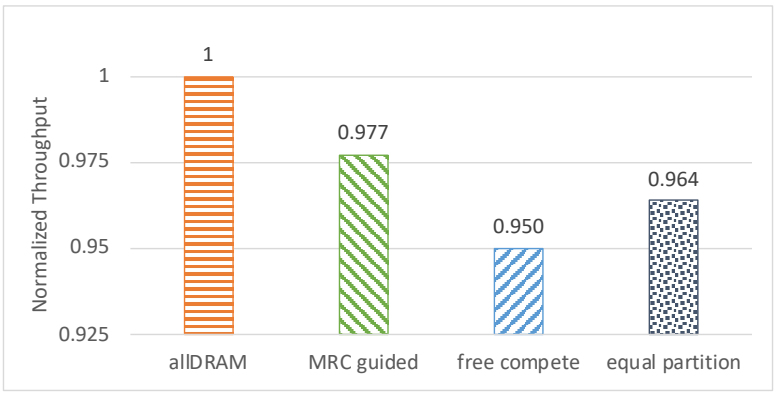


Figure 5.6: Normalized throughput of 4 tenants with fast-tier memory partitioning in YCSB workloads

5.2.4.3 Phase-Changing Case Study

In this case study, we construct a synthetic workload comprising two distinct phases. Both phases follow a Zipfian distribution with an *alpha* value of 0.9, and each phase includes 500 million references, totaling 1000 million accesses. The workload consists of 50 million distinct objects. In the first phase, hot items are concentrated on keys with smaller values, meaning keys with smaller values have a higher access probability. In the second phase, the hot items shift to keys with larger values. Consequently, the hot item set changes as the phase transitions. Figure 5.7 presents the CDF of access frequency for all keys throughout the entire trace, illustrating that the hot keys are distributed at the two ends of the key space.

In sdTMM configuration, the size of the fast memory tier is set to 50% of the working set size. We use one cache stressor to send the references. In two sets of evaluations,

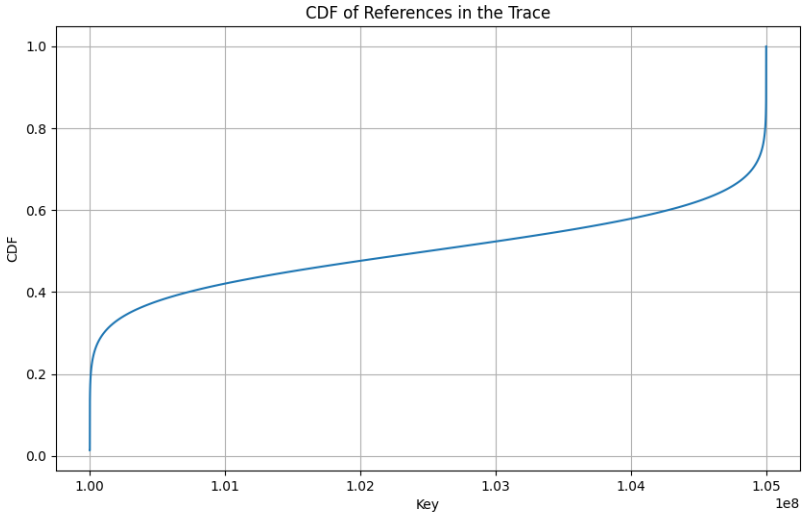
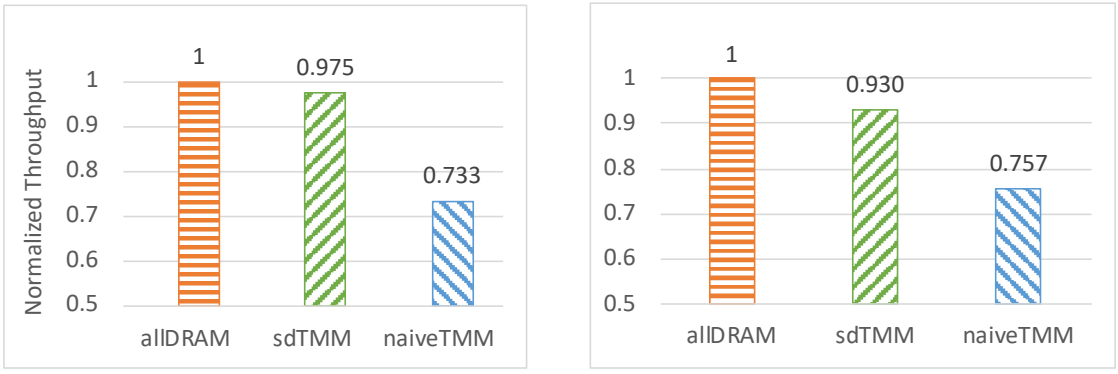


Figure 5.7: CDF of access frequency of a synthetic phase-changing workload

the value size of all references is uniformly 300 B (small object size) and 4 KB (large object size), respectively.

Figure 5.8 (a) shows, in small object size, the normalized throughput compared to the allDRAM single-tier system. With 50% of the local DRAM replaced by slow memory, sdTMM exhibits a 2.5% performance impact, whereas naiveTMM shows a 26.7% degradation. At the end of phase 1, sdTMM and naiveTMM achieve a fast-tier hit rate of 90.2% and 87.15%, respectively. At the end of phase 2, which also marks the end of the entire trace, the fast-tier hit rate of sdTMM, and naiveTMM are 89.76%, and 87.27%, respectively. Throughout the entire trace, sdTMM promoted only about 1% of the items compared to naiveTMM, yet achieved a higher fast-tier hit rate. This demonstrates that sdTMM effectively detects and maintains hot objects in



(a) Small Obj Size

(b) Large Obj Size

Figure 5.8: Normalized throughput of synthetic phase-changing workload

the fast tier during phase 1. As the workload transitions to phase 2 and the hot items shift entirely to large keys in the slow tier, sdTMM successfully detects this change and migrates the new hot items to the fast tier. Consequently, sdTMM maintains a fast-tier hit rate close to 90%, outperforming naiveTMM with significantly fewer promotions.

Figure 5.8 (b) shows the result when references are in large object size, with slightly decreased performance compared to the result with small object size. sdTMM and naiveTMM exhibit performance degradation of 7.0% and 24.3%, respectively. This is attributed to the increased moving bandwidth required for migrating large object-sized items between memory tiers, which impacts system throughput.

5.2.4.4 Phase-Changing & Memory Partitioning Case Study

Finally, we evaluate the performance of sdTMM in identifying and migrating hot items, as well as in partitioning fast-tier memory, using two tenants and phase-changing YCSB workloads with different *alpha* values.

Tenant 1 runs a master phase-changing workload with the first 100 million references from YCSB ($alpha = 0.8$), followed by a second 100 million references from YCSB ($alpha = 1.2$). Tenant 2 runs a master phase-changing workload starting with 100 million accesses from YCSB ($alpha = 1.2$), followed by 100 million accesses from YCSB ($alpha = 0.8$). Thus, in this multi-tenant scenario, both tenants run workloads that feature hot item set phase changes. In the second phase, the MRC-guided fast-tier memory partitioning decisions will differ from those in phase 1, specifically reversing the tenant memory distribution. sdTMM needs to adjust the tenants' memory sizes to achieve the desired allocation changes, supported by the background resizing worker. The fast-tier memory size for sdTMM is set to 25% of the working set size.

Figure 5.9 shows the results. With 75% of fast DRAM replaced by slow-tier memory, the throughput degradation compared to the all-DRAM system for MRC-guided partition, free competition, and equal partition strategies are 4.3%, 7.5%, and 9.8%, respectively.

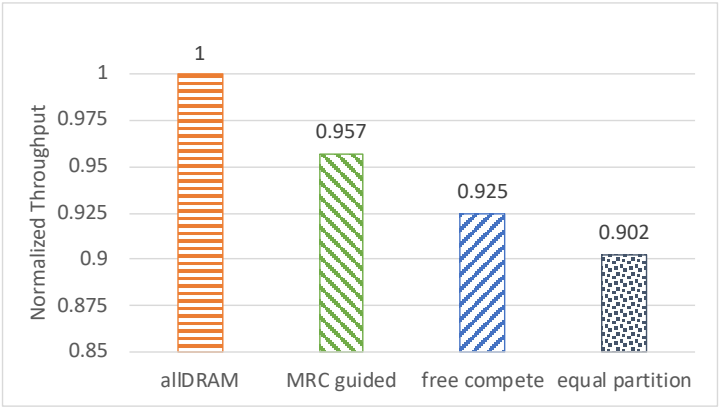


Figure 5.9: Normalized throughput of 2 tenants with phase-changing YCSB workloads

At the end of phase 1, the fast-tier hit rate of sdTMM with MRC-guided partition exceeds 80%. At this point, the fast-tier memory partition between Tenant 1 and Tenant 2 is 0.96 vs. 0.04. As the cache execution transitions into phase 2, the tenants' access patterns change. Accordingly, the fast-tier memory partition updates to 0.04 vs. 0.96 for the two tenants. The background resizing worker detects that Tenant 1 is overusing memory while Tenant 2 requires more space, prompting it to gradually shrink Tenant 1's memory space and expand Tenant 2's memory space.

At the end of phase 2, the fast-tier hit rate of sdTMM with MRC-guided partition reaches 77.64%, compared to 94.94%, 73.84%, and 69.37% for all-DRAM, free competition, and equal partition strategies, respectively. Overall, sdTMM with MRC-guided partition demonstrates the lowest performance impact, despite having only 25% of the working set size in fast DRAM memory.

5.2.5 Impact of Fast-Tier Memory Size

In this section, we analyze the impact of fast-tier memory size on sdTMM's performance. Specifically, we consider the allocation efficiency for workloads characterized by a highly skewed distribution of hot items. The fast-tier size should ideally be just sufficient to hold the hot set, while other items can be placed in slow-tier memory. This approach avoids wasting limited DRAM resources and allows sdTMM to dynamically manage item placement as needed.

To demonstrate the impact of fast-tier size, we use a master workload composed of phase-changing Twitter cluster traces. This master workload concatenates Twitter cluster52.0 and cluster54.0 traces, both of which exhibit high skewness with *alpha* values of 1.6 and 1.2, respectively. Figure 5.10 shows the MRC for these two Twitter cluster traces.

In the first sdTMM configuration, we set the fast-tier size to 50% of the working set size, which is large enough to hold the hot items of both Twitter cluster traces. We compare sdTMM's performance against sdTMM with only background demotions, referred to as sdTMM-demotion, where no items are promoted from the slow tier to the fast tier. From Figure 5.11a, we observe no significant performance difference between sdTMM and sdTMM-demotion. The performance degradation of the two

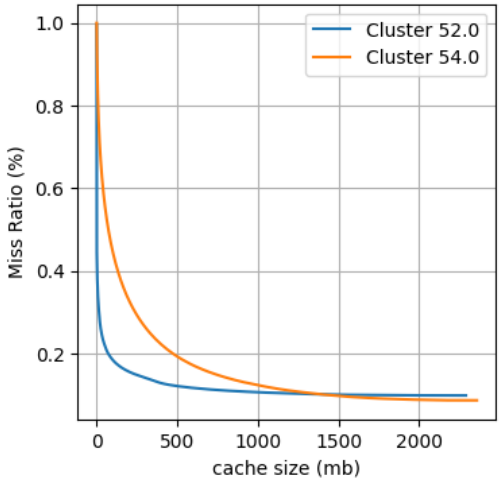
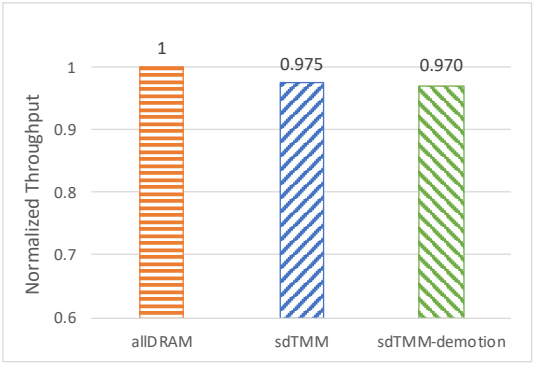


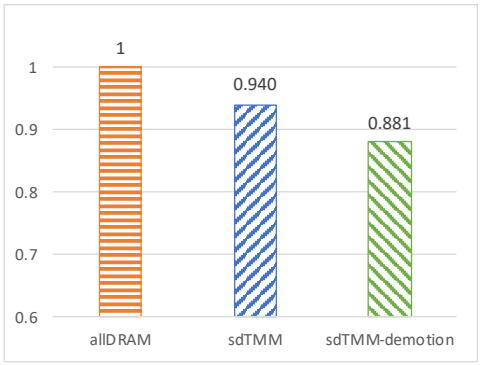
Figure 5.10: MRCs of Twitter Cluster52.0 and Cluster54.0

versions compared to all-DRAM is 2.5% and 3.0%, respectively, with fast-tier hit rates of 82.55% and 80.62%, respectively. Despite lacking hot item identification and promotion, sdTMM-demotion shows similar performance due to the large fast-tier memory size setting.

In the second sdTMM configuration, we set the fast-tier size to 20% of the working set size, meaning the hot set from both phases cannot fit entirely in the fast tier. Figure 5.11b shows that the degradation of sdTMM and sdTMM-demotion compared to all-DRAM is 6.0% and 11.9%, respectively, with fast-tier hit rates of 75.70% and 71.28%, respectively. The hot item identification and promotion in sdTMM contribute to the increased fast-tier hit rate and throughput.



(a) Large fast-tier size



(b) Small fast-tier size

Figure 5.11: Normalized throughput of Twitter phase-changing workload

5.2.6 Tail Latency

sdTMM has been shown to have a low impact on system performance, even when a significant portion of fast DRAM is replaced with shared slow memory. Our next objective is to determine if item hotness tracking, data migration, fast-tier memory allocation and re-sizing disproportionately affect system latency.

We evaluate the tail latency of sdTMM using the case discussed in Section 5.2.4.4. Table 5.1 presents the results for allDRAM, sdTMM, and allSLOW (where all memory is bound to the shared slow memory). We see that by replacing 75% of fast DRAM with shared slow memory, its GET and SET latencies are higher than allDRAM but lower than allSLOW.

This study case includes combined scenarios of phase-changing and dramatic changes

in fast-tier memory partitioning. At the end of phase 1 and phase 2, sdTMM's fast-tier hit rates are 80.03% and 77.64%, respectively, compared to 92% and 94.88% for allDRAM. The fast-tier hit rate impacts sdTMM's GET tail latency compared to the allDRAM system.

Furthermore, at the start of phase 2, the fast-tier memory distribution shifts dramatically to an opposite distribution, prompting the memory re-sizer to adjust slab allocations among tenants. This also affects sdTMM's SET tail latency.

This shows sdTMM's disadvantage in tail latency when using a shared slow memory tier under strict fast-memory size constraints. For latency-sensitive applications, we recommend either utilizing a larger fast-memory configuration or avoiding deployment in a multi-tenant environment.

Table 5.1
Tail Latency Measurements

Type	**90th**	**99th**	**99.9th**
GET tail latency (ns)			
allDRAM	1046	1963	2834
sdTMM	2069	2806	4109
allSLOW	2464	3459	6190
SET tail latency (ns)			
allDRAM	1179	5265	11380
sdTMM	2211	7823	16964
allSLOW	2655	12284	19007

5.2.7 Space and Time Cost

In this section, we evaluate the time and space overhead of sdTMM. As discussed in Section 5.1.5, sdTMM memory partitioning can be performed either through online reference sampling and MRC construction or by following offline memory allocation decisions. For the case of online profiling and modeling, Section 4.2.7 provided a detailed measurement of the space and time overhead of tenant MRC modeling with the KRR model.

Additionally, there is a time and space cost associated with sdTMM related to hotness identification: item reuse time tracking. In the fast memory tier, reuse time is accumulated for each item re-access within a rolling window, using the mean as the hotness threshold. Note that we do not record or update an item's reuse time upon re-access in fast-tier for efficiency purposes. In the slow memory tier, each item's reuse time is recorded along with the last access time in the metadata upon re-access. We evaluate the running time for three variants of sdTMM against the 400 million YCSB multi-tenant workloads used in Section 5.2.4.4. The results are shown in Table 5.2. In the first variant of sdTMM, reuse time is not tracked for either memory tier. The second variant tracks only the window mean reuse time in the fast memory tier. The third variant tracks fast-tier window reuse time and updates the item's reuse time on each re-access in the slow memory tier. We observe similar time costs in the first two

sdTMM variants. And the time cost for tracking and updating slow-tier item reuse time is negligible.

Table 5.2
Running Time Comparison for Processing 400 Million YCSB multi-tenant Requests

Reuse Time Tracking Efficiency	
Methods	**Time (sec)**
No reuse time tracking	448.73
Fast-tier win mean reuse time tracking only	451.67
Fast & slow tier reuse time tracking	452.04

However, we increased the item's metadata by 4 bytes to store the item's reuse time in the LRU list implementation. But this increased space cost only affects items in the slow memory tier since we do not need to store the reuse time for fast-tier items. Thus the space overhead of the reuse time tracking is proportional to the number of objects stored in the slow memory tier. The estimated percentage of space overhead is *4 bytes * R / average k-v size*, where R is the memory size ratio of the slow-tier. For instance, in the case discussed in Section 5.2.4.4, $R = 0.75$, and the average size of k-v items is 324 bytes, then the space overhead is 0.9% of the working set size.

5.3 Chapter Summary

In this chapter, we present the design and implementation of sdTMM, a software-defined multi-tier memory management system designed for multi-tenant environments. sdTMM is capable of detecting application-level item hotness dynamics across two memory tiers and optimizing item placement with efficient background item migrations. Evaluations across various workloads demonstrate that our approach, even with 80% of the fast memory replaced by CXL-shared slow memory, incurs an average performance impact of 13%, with a best-case performance impact as low as 2.2% compared to an all-fast memory over-provisioned system. Comparison against naiveTMM show that sdTMM achieves a higher fast-tier hit rate with significantly fewer data migrations. Evaluations in multi-tenant environments indicate that sdTMM, equipped with MRC-guided memory partitioning, outperforms other allocation strategies. We also assess the overhead and tail latency of our approach and discuss its limitations.

Chapter 6

Conclusion

In today's multi-level storage architectures, in-memory k-v stores play a crucial role in ensuring low-latency system performance. Retrieving objects from a k-v store system deployed in memory on a front-end server is significantly faster than accessing them from a remote back-end server. To optimize system performance, it is essential to innovate and integrate techniques such as effective cache replacement algorithms, efficient memory allocation in multi-tenant environments, and expanding memory capacity with support from advanced tiered memory management systems. This dissertation proposes an integrative scheme to address these challenges, showing promising results through extensive experimentation. This Chapter summarizes our contributions and discusses future work.

6.1 Contributions

Our main contributions are as follows.

1. **Dynamic LRU Configuration for Redis (DLRU)**: We conducted an in-depth analysis of the K-LRU behavior in Redis, identifying potential variations in miss ratios based on different K values. By introducing a dynamic configuration scheme for K, which leverages a low-overhead miniature cache simulator and a cost model, we can predict miss ratios and optimize performance trade-offs. This dynamic approach enhances Redis throughput by up to 32.5% over the default static K setting.

2. **Locality- and Latency-Aware Memory Partitioning (kRedis)**: Extending the exploration of K-LRU, we developed kRedis, which operates in a multi-tenant k-v store environment. This system includes a locality- and latency-aware memory partitioning scheme that dynamically allocates memory based on the specific locality and latency characteristics of each tenant. The evaluation results show substantial performance improvements, with kRedis reducing average access latency by up to 50.2% and increasing throughput by up to 262.8% compared to standard Redis. Additionally, kRedis outperforms a state-of-the-art memory allocation design, with improvements of up to 24.8% in average

access latency and 61.8% in throughput.

3. **Software-Defined Tiered Memory Management (sdTMM)**: Drawing on the capabilities of emerging Compute Express Link (CXL) memory-sharing technologies, we designed sdTMM, a software-defined tiered memory management system. This system integrates fast local DRAM with slower but larger CXL-shared memory to create an efficient multi-tier memory pool. By dynamically identifying and placing hot data, efficiently migrating items among memory tiers based on their popularity, and implementing locality-aware multi-tenant memory partitioning, sdTMM optimizes memory utilization while maintaining high performance. Our evaluations indicate that sdTMM, even with 80% of fast memory replaced by CXL-shared slow memory, incurs an average performance impact of 13%, and the best-case of only 2.2% compared to an all-fast memory over-provisioned system.

6.2 Future Work

We plan to conduct an ablation study to evaluate the individual contributions of optimizations made in the current sdTMM design, including avoidance of cascade evictions, new promotion threshold, proactive memory collection, and customized insertion points for fast and slow tiers. By systematically disabling each optimization

and observing the impact on system performance, we aim to understand their individual and combined effects. This study will help identify the most effective strategies and guide future improvements to sdTMM.

Furthermore, we will integrate the random sampling-based LRU eviction policy into Cachelib and sdTMM, and investigate its benefit against current approximate LRU implementations, as well as other policies supported in Cachelib.

We will also investigate other random sampling policies using diverse metrics, such as access frequency and object expiration time, as priority functions. These policies will be utilized in memory management for broader usage scenarios.

Additionally, future work will explore further optimizations and potential expansions of these techniques to other areas of memory management and storage systems.

References

[1] Y. Wang, J. Yang, and Z. Wang, "Dynamically configuring lru replacement policy in redis," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '20. New York, NY, USA: Association for Computing Machinery, 2021, p. 272–280.

[2] Y. Wang, J. Yang, Z. Wang, and Wang, "Multi-tenant in-memory key-value cache partitioning using efficient random sampling-based lru model," *IEEE Transactions on Cloud Computing*, vol. 11, no. 4, pp. 3601–3618, 2023.

[3] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "Hotring: A hotspot-aware in-memory key-value store," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 239–252.

[4] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in

2015 USENIX Annual Technical Conference (USENIX ATC 15). Santa Clara, CA: USENIX Association, 2015, pp. 57–69.

[5] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 499–511.

[6] D. Byrne, N. Onder, and Z. Wang, "Faster slab reassignment in memcached," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 353–362.

[7] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, 2015, pp. 95–110.

[8] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data, "Characterizing storage workloads with counter stacks," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 2014, pp. 335–349.

[9] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.

[10] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 423–432.

[11] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: Dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys '18. New York, NY, USA: Association for Computing Machinery, 2018.

[12] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 103–116.

[13] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS.* IEEE, 2004, pp. 20–27.

[14] X. Xiang, C. Ding, H. Luo, and B. Bao, "Hotl: a higher order theory of locality," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1. ACM, 2013, pp. 343–356.

[15] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 55–65.

[16] memcached. (2020, May) memcached. [Online]. Available: https://memcached.org

[17] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. San Francisco, CA: USENIX Association, Mar. 2003.

[18] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USA: USENIX Association, 2001, p. 91–104.

[19] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with CACHEUS," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, Feb. 2021, pp. 341–354.

[20] Redis. (2019) Redis replacement policy. [Online]. Available: https://redis.io/topics/lru-cache

[21] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *15th USENIX Symposium on Networked Systems Design*

and Implementation (NSDI 18). Renton, WA: USENIX Association, Apr. 2018, pp. 389–403.

[22] SNIA. (2020) Msr cambridge traces. Accessed: 2020-03-15. [Online]. Available: http://iotta.snia.org/traces/388

[23] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, p. 60–71, Jun. 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815971

[24] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM System Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[25] J. Yang, Y. Wang, and Z. Wang, "Efficient modeling of random sampling-based lru," in *50th International Conference on Parallel Processing*, ser. ICPP 2021. New York, NY, USA: Association for Computing Machinery, 2021.

[26] J. Yang, Y. 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)*. USENIX Association, Nov. 2020, pp. 191–208.

[27] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger, "The cachelib caching engine: Design and experiences at scale," in *14th USENIX Symposium*

on Operating Systems Design and Implementation (OSDI 20). USENIX Association, Nov. 2020, pp. 753–768.

[28] C. Pan, X. Hu, L. Zhou, Y. Luo, X. Wang, and Z. Wang, "Pace: Penalty aware cache modeling with enhanced aet," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, ser. APSys '18. New York, NY, USA: Association for Computing Machinery, 2018.

[29] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 19–19.

[30] A. Cidon, A. Eisenman, and M. Alizadeh, "Cliffhanger: Scaling performance cliffs in web memory caches," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 379–392.

[31] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: a dynamic multi-tenant key-value cache," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 321–334.

[32] C. Pan, X. Wang, Y. Luo, and Z. Wang, "Penalty- and locality-aware memory

allocation in redis using enhanced aet," *ACM Trans. Storage*, vol. 17, no. 2, may 2021.

[33] D. Byrne, N. Onder, and Z. Wang, "mPart: Miss-ratio curve guided partitioning in key-value stores," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ser. ISMM 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 84–95.

[34] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye, "Fast miss ratio curve modeling for storage cache," *ACM Trans. Storage*, vol. 14, no. 2, pp. 12:1–12:34, Apr. 2018.

[35] G. Quan, J. Tan, A. Eryilmaz, and N. Shroff, "A new flexible multi-flow lru cache management paradigm for minimizing misses," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, no. 2, jun 2019. [Online]. Available: https://doi.org/10.1145/3341617.3326154

[36] D. S. Berger, B. Berg, T. Zhu, M. Harchol-Balter, and S. Sen, "Robinhood: tail latency-aware caching—dynamically reallocating from cache-rich to cache-poor," in *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'18. USA: USENIX Association, 2018, p. 195–212.

[37] J. Choi, S. Blagodurov, and H.-W. Tseng, "Dancing in the dark: Profiling for tiered memory," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 13–22.

[38] D.-s. Byeon, S.-s. Lee, Y.-h. Lim, D. Kang, W.-k. Han, D.-h. Kim, and K.-d. Suh, "A comparison between 63nm 8gb and 90nm 4gb multi-level cell nand flash memory for mass storage application," in *2005 IEEE Asian Solid-State Circuits Conference*, 2005, pp. 13–16.

[39] W. Chien, Y. Ho, H. Cheng, M. BrightSky, C. Chen, C. Yeh, T. Chen, W. Kim, S. Kim, J. Wu, A. Ray, R. Bruce, Y. Zhu, H. Ho, H. Lung, and C. Lam, "A novel self-converging write scheme for 2-bits/cell phase change memory for storage class memory (scm) application," in *2015 Symposium on VLSI Technology (VLSI Technology)*, 2015, pp. T100–T101.

[40] Y. Halawani, B. Mohammad, D. Homouz, M. Al-Qutayri, and H. Saleh, "Modeling and optimization of memristor and stt-ram-based memory for low-power applications," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 3, pp. 1003–1014, 2016.

[41] A. Driskill-Smith, S. Watts, V. Nikitin, D. Apalkov, D. Druist, R. Kawakami, X. Tang, X. Luo, A. Ong, and E. Chen, "Non-volatile spin-transfer torque ram (stt-ram): Data, analysis and design requirements for thermal stability," in *2010 Symposium on VLSI Technology*, 2010, pp. 51–52.

[42] F. T. Hady, A. Foong, B. Veal, and D. Williams, "Platform storage performance with 3d xpoint technology," *Proceedings of the IEEE*, vol. 105, no. 9, pp. 1822–1833, 2017.

[43] V. Mironov, I. Chernykh, I. Kulikov, A. Moskovsky, E. Epifanovsky, and A. Kudryavtsev, "Performance evaluation of the intel optane dc memory with scientific benchmarks," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, 2019, pp. 1–6.

[44] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019.

[45] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenter," *SIGARCH Comput. Archit. News*, vol. 45, no. 2, p. 521–534, jun 2017.

[46] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 331–345.

[47] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, ser. EuroSys '16. New York, NY, USA: Association for Computing Machinery, 2016.

[48] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 392–407.

[49] L. Liu, Y. Li, Z. Cui, Y. Bao, M. Chen, and C. Wu, "Going vertical in memory management: Handling multiplicity by multi-policy," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 169–180.

[50] L. Liu, Y. Li, C. Ding, H. Yang, and C. Wu, "Rethinking memory management in modern operating system: Horizontal, vertical or random?" *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1921–1935, 2016.

[51] X. Zhang, S. Dwarkadas, and K. Shen, "Towards practical page coloring-based multicore cache management," in *Proceedings of the 4th ACM European Conference on Computer Systems*, ser. EuroSys '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 89–102.

[52] L. Liu, S. Yang, L. Peng, and X. Li, "Hierarchical hybrid memory management in os for tiered memory systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2223–2236, 2019.

[53] M. B. Olson, B. Kammerdiener, M. R. Jantz, K. A. Doshi, and T. Jones, "Portable application guidance for complex memory systems," in *Proceedings*

of the International Symposium on Memory Systems, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 156–166.

[54] J. Choi, S. Blagodurov, and H.-W. Tseng, "Dancing in the dark: Profiling for tiered memory," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 13–22.

[55] S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang, "vtmm: Tiered memory management for virtual machines," in *Proceedings of the Eighteenth European Conference on Computer Systems*, ser. EuroSys '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 283–297. [Online]. Available: https://doi.org/10.1145/3552326.3587449

[56] A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 237–248.

[57] Y. Du, M. Zhou, B. R. Childers, D. Mossé, and R. Melhem, "Supporting superpages in non-contiguous physical memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015, pp. 223–234.

[58] G. Cox and A. Bhattacharjee, "Efficient address translation for architectures with multiple page sizes," *SIGPLAN Not.*, vol. 52, no. 4, p. 435–448, apr 2017.

[59] J. Gandhi, V. Karakostas, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. S. Ünsal, "Range translations for fast virtual memory," *IEEE Micro*, vol. 36, no. 3, pp. 118–126, 2016.

[60] F. X. Lin and X. Liu, "Memif: Towards programming heterogeneous memory asynchronously," *SIGARCH Comput. Archit. News*, vol. 44, no. 2, p. 369–383, mar 2016.

[61] C. Consortium. (2023, Jan) Compute express link™: The breakthrough cpu-to-device interconnect. [Online]. Available: https://www.computeexpresslink.org/

[62] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 45–51.

[63] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 742–755. [Online]. Available: https://doi.org/10.1145/3582016.3582063

[64] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal, M. D. Hill, M. Fontoura, and R. Bianchini, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 574–587.

[65] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic, B. Morris, C. Mukherjee, J. Ren, G. Thelen, P. Turner, C. Villavieja, P. Ranganathan, and A. Vahdat, "Towards an adaptable systems architecture for memory tiering at warehouse-scale," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 727–741. [Online]. Available: https://doi.org/10.1145/3582016.3582031

[66] K. Lee, S. Kim, J. Lee, D. Moon, R. Kim, H. Kim, H. Ji, Y. Mun, and Y. Joo, "Improving key-value cache performance with heterogeneous memory tiering: A case study of cxl-based memory expansion," *IEEE Micro*, pp. 1–11, 2024.

[67] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 487–498.

[68] Redis. (2019) Redis 4.0.13. [Online]. Available: http://download.redis.io/releases/

[69] Mutilate. (2019) Mutilate. [Online]. Available: https://github.com/leverich/mutilate

[70] C. Pan, Y. Luo, X. Wang, and Z. Wang, "pRedis: Penalty and locality aware memory allocation in redis," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 193–205.

[71] Redis. (2021) Hiredis. [Online]. Available: https://github.com/redis/hiredis

[72] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, nov 2008. [Online]. Available: https://doi.org/10.1145/1416944.1416949

[73] Twitter. (2020, Dec.) Twitter cache trace. [Online]. Available: https://github.com/twitter/cache-trace

[74] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, "Fifo queues are all you need for cache eviction," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 130–149. [Online]. Available: https://doi.org/10.1145/3600006.3613147

[75] Intel. (2024, March) Intel multi-tier cachelib. [Online]. Available: https://github.com/intel/CacheLib

[76] Meta. (2024, May) Cachelib - pluggable caching engine to build and scale high performance cache services. [Online]. Available: https://cacheLib.org

[77] S. L. Andi Kleen. (2024, May) numactl. [Online]. Available: https://github.com/numactl/numactl

[78] S. McAllister, B. Berg, J. Tutuncu-Macias, J. Yang, S. Gunasekar, J. Lu, D. S. Berger, N. Beckmann, and G. R. Ganger, "Kangaroo: Caching billions of tiny objects on flash," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 243–262. [Online]. Available: https://doi.org/10.1145/3477132.3483568

[79] K. Lee, S. Kim, J. Lee, D. Moon, R. Kim, H. Kim, H. Ji, Y. Mun, and Y. Joo, "Improving key-value cache performance with heterogeneous memory tiering: A case study of cxl-based memory expansion," *IEEE Micro*, pp. 1–11, 2024.

[80] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[81] Intel. (2024, May) Intel® vtune™ profiler - find and fix performance bottlenecks quickly and realize all the value of your hardware. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[82] R. Labs. (2020, May) redis. [Online]. Available: https://redis.io

[83] F. Olken, "Efficient methods for calculating the success function of fixed-space replacement policies," Lawrence Berkeley Lab., CA (USA), Tech. Rep., 1981.

[84] A. Lagar-Cavilla, J. Ahn, S. Souhlal, N. Agarwal, R. Burny, S. Butt, J. Chang, A. Chaugule, N. Deng, J. Shahid, G. Thelen, K. A. Yurtsever, Y. Zhao, and P. Ranganathan, "Software-defined far memory in warehouse-scale computers," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 317–330. [Online]. Available: https://doi.org/10.1145/3297858.3304053