The Australian National University
2600 ACT | Canberra | Australia

School of Computing

College of Engineering,
Computing & Cybernetics
(CECC)

HyperCache: A High Capacity Cache for Small Objects over Hybrid Memory

— 24 pt Honours project (S2/S1 2022–2023)

A thesis submitted for the degree
Bachelor of Advanced Computing (Honours)

By:
Junming Zhao

Supervisor:
Dr. Shoaib Akram

January 2025

Declaration:

I declare that this work:

- upholds the principles of academic integrity, as defined in the University Academic Misconduct Rules;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or Wattle site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

January, Junming Zhao

Acknowledgements

I want to express my sincere gratitude to my supervisor, Dr. Shoaib Akram, for his invaluable guidance and support throughout my honours project. I also thank the VICS research team for their assistance and encouragement. Additionally, I would like to express my appreciation for the advancement in AI tools, such as ChatGPT and BARD, which have helped me accelerate tedious tasks such as Bash scripting for automating commands and Python scripting for data visualization. Lastly, I am deeply grateful to my family for their constant and unwavering support. Thank you all for your help and support.

Abstract

Caching is essential for improving service availability and efficiency in data centers. Cache stores frequently used user data in a faster, more expensive tier. Caching tier resides close to users. It is, therefore, paramount that its utility is maximized to avoid sending requests over the wide area network. In many online services today, caches are backed by main memory. Furthermore, most data centers today use DRAM (Dynamic Random-Access Memory) as the main memory. However, DRAM's capacity is under pressure due to technology scaling limitations. This trend in memory technology poses a challenge in accommodating a growing user base producing data that must be cached, requiring high-capacity memory technologies.

In this thesis, we investigate the case of a hybrid cache that exploits DRAM and emerging PMem (Persistent Memory). We propose a new hybrid cache architecture, *HyperCache*, which combines DRAM's high speed with PMem's high capacity. The key features of our design include:

- Extending state-of-the-art in-memory cache, namely *Segcache*, to use both DRAM and PMem to improve efficiency and economy;
- Efficient group-based machine learning eviction algorithms over DRAM and PMem;
- An optional optimization to divide the in-memory hash table for fast look-ups into smaller parts and store parts of it in PMem.

Our evaluations demonstrate that HyperCache outperforms Segcache in the following ways:

- HyperCache allows a reduction in DRAM usage by 70%, with only a 20% loss of performance in throughput.
- HyperCache improves the hit ratio of real-world data-center request traces by 4.5% for the same DRAM usage.

HyperCache provides a more efficient, high-capacity, and cost-effective cache solution for data centers. This thesis also provides new insights into the design of data center caches, presenting a potential path toward more efficient and scalable caching solutions.

List of Abbreviations

- API: Application Programming Interface
- ARC: Adaptive Replacement Cache
- CDN: Content Distributed Network
- CPU: Central Processing Unit
- CTE: Closest-To-Expire
- DDoS: Distributed Denial-of-Service
- DDR5: Double Data Rate 5
- DRAM: Dynamic Random-Access Memory
- FIFO: First-In-First-Out
- HDD: Hard Disk Drive
- HTBL: Hashtable
- KV: Key-Value
- LFB: Relaxed Baledy Algorithm
- LFU: Least-Frequently-Used
- LRU: Least-Recently-Used
- LR: Linear Regression
- ML: Machine Learning
- MQPS: Million Requests Per Second
- MSR: Microsoft Research Cambridge
- NN: Neural Network
- NVM: Non-Volatile Memory

- NVMM: Non-Volatile Main Memory
- NVMe: Non-Volatile Memory Express
- OS: Operating System
- PCIe: Peripheral Component Interconnect Express
- PMem: Persistent Memory
- RL: Reinforcement Learning
- SATA: Serial AT Attachment
- SLRU: Segmented Least Frequently Used
- SSD: Solid-State Drive
- TTL: Time-To-Live
- XGB: eXtreme Gradient Boost

Table of Contents

Chapter 1

Introduction

1.1 Overview and Motivation

In this chapter, we provide a comprehensive overview and motivation behind this study. It discusses the importance of high-capacity data center caches and the potential of Persistent Memory (PMem). We also provide a brief overview of the problem statement, the objectives, and the research contribution.

1.1.1 Caches

In the era of digitization, data centers are the engines supporting a large number of services, from social media to financial systems. Central to their performance is the efficient management and utilization of vast and exponentially growing volumes of data. The challenge of high-speed large-volume data processing is becoming increasingly important.

This is where data center caching steps in: Caching works by storing frequently accessed data in a high-speed memory, such as Dynamic Random-Access Memory (DRAM), while the rest of the data are stored in Hard Disk Drives (HDDs) storage. This allows applications to access the data more quickly, which gives significant improvements in performance and user experience.

1.1.2 The Potential of PMem

The emergence of PMem has introduced a compelling alternative to traditional memory types. PMem surpasses traditional memory devices like flash drives and Solid-State Drives (SSDs), by offering speeds comparable to DRAM. Moreover, PMem holds a unique advantage over DRAM due to its lower cost and its non-volatile nature (that PMem can retain data without power). PMem's unique combination of advantages makes it a strong candidate as a storage medium for data center cache.

Because PMem is a relatively new technology, its potential in the context of caching remains largely unexplored, particularly when considering hybrid caching that integrates PMem with DRAM. Most of the existing data center cache systems are designed for DRAM only and do not yet consider PMem. A DRAM-PMem hybrid approach, equipped with efficient and effective eviction policies, could unlock considerably many benefits, such as larger caching capacity, higher caching performance, better cost-efficiency, lower energy consumption, etc.

1.1.3 Group-based ML Cache Eviction

Group-based Eviction

The cache eviction policy of a system determines which data should be removed from the cache to make room for new data. Such policy is a critical factor in the effectiveness of a caching system. Traditional heuristic approaches, such as Least Recently Used (LRU) or First-In First-Out (FIFO), operate on individual data objects. However, while PMem's performance exceeds traditional storage options like SSD, there's still a significant gap between PMem and DRAM [1]. This implies that the traditional methods may not be optimal for PMem systems due to their higher read/write latency.

Group-based eviction offers a potential solution to this issue. Twitter's state-of-art caching design *Segcache* [2] employs a Time-To-Lived(TTL)-based object grouping method which has demonstrated significant performance improvement. By considering related data objects as a group, eviction decisions and processes can be executed collectively, greatly reducing the costly read/write operations and thereby enhancing the system's performance. This approach can be particularly beneficial when it comes to a hybrid caching system that uses PMem.

ML Eviction

However, determining which group should be evicted is a complex decision problem. Traditional heuristic policies, which were designed for individual objects, may not be suitable for group-based decision-making. Moreover, these heuristic approaches lack the ability to adapt to dynamic workloads and access patterns, as pointed out by [3].

On the other hand, machine learning approaches can overcome these limitations. By learning historical data, the ML model can predict future accesses based on current group information. In essence, group-based ML eviction strategies could lead to more intelligent, adaptive, and efficient caching systems, compared to traditional heuristic policies.

1.2 Problem Statement

Problem 1: Adapting In-Memory Designs for PMem Characteristics

PMem has different physical characteristics from DRAM, such as writing speed and durability. Most current in-memory[1] caching designs are optimized for DRAM and do not consider these characteristics. Therefore, the existing designs may not be efficient with PMem, thus posing a challenge in the adoption of PMem into the new model.

Problem 2: Efficient Strategies for Managing Hybrid Memory Layers

The management across DRAM and PMem in a hybrid caching system represents another significant challenge. Traditionally, caching systems are designed for a homogeneous memory environment, lacking support for multiple layers of different memory types.

Therefore, extending an in-memory caching system into a hybrid mode necessitates the development of novel strategies, algorithms, and potentially a fundamental redesign to effectively leverage the advantages of both memory types.

Problem 3: Group-Based ML Eviction in Hybrid Caching System

As previously mentioned, Group-based ML eviction strategies have the potential to improve eviction management in a hybrid DRAM-PMem system. However, the various steps in these strategies, including data collection, training, and forecasting, will inevitably impose additional loads on the system. Finding a balance between system overhead and eviction effectiveness is a considerable challenge.

1.3 Our Objectives and Contribution

Driven by the above challenges, our work extends *Segcache* [2], a current state-of-the-art in-memory caching system, to adopt a hybrid approach that integrates both DRAM and PMem. We aim to explore the benefits of this hybrid caching system, develop group-based ML eviction algorithms suitable for the system, and evaluate the system's performance.

The principal contributions of our work are as follows:

- Modify Segcache to a combination of DRAM and PMem, making the caching system more flexible and adaptable
- Implement and Evaluate different group-based eviction algorithms powered by Machine Learning
- Conduct a thorough performance analysis of the proposed hybrid system, comparing it with the traditional DRAM-only system

[1]In-memory: In our context, in-memory is equivalent as DRAM only

- Demonstrate the improved performance and cost-efficiency of the proposed hybrid caching system

While our study focuses on the DRAM-PMem hybrid model within the context of Segcache, the methodologies and insights derived from our research could also apply to other hybrid configurations, such as DRAM-SSD. By providing empirical evidence, as well as establishing generalizable principles and novel strategies for hybrid caching systems, this work has the potential to contribute substantially to the academic discourse in this under-studied yet increasingly important area.

1.4 Thesis Outline

In the next chapter, we provide background on data center caching, including key challenges, and we also discuss memory and storage technology trends. We discuss Segcache, on which we base our research. Then, after discussing background, in the subsequent chapter (Chapter 3), we discuss related work, including application of emerging memories, and machine learning-based cache eviction approaches. We precede the design and implementation details with experimental methodology. We discuss methodology first because our key design decisions are informed by quantitative analysis of cache behavior for realistic data center workloads. We then discuss design and implementation details (Chapter 5), followed by evaluation results (Chapter 6). We conclude this thesis by discussing promising directions for future work.

Chapter 2

Background

This chapter discusses the essential background related to data center caching. We provide an introduction to the concept of data center caching, its importance, the challenges it currently faces, and the core component of caching: eviction policies. Next, we explore the memory technologies applicable in caching, specifically DRAM, PMem, and SSD. We then go through a detailed explanation of Segcache, the first TTL-indexed cache in literature, since our work is built on this system. Lastly, we briefly overview machine learning with its basic concepts.

2.1 Data Center Caching Overview

2.1.1 Introduction to Caching

A caching system in data center is a high-speed storage component, coupled with management algorithms, that is designed to store frequently accessed data to reduce data center latency. It is often placed in front of a larger, slower back-end database hosted on disks or servers. The cache itself is typically implemented using faster memory, such as DRAM, that provides significantly faster retrievals.

A cache management algorithm refers to a set of strategies and techniques employed to determine how data is stored, organized, and replaced within a cache. One key component is the eviction policy: it determines which data should be retained or evicted (i.e. kept or removed) from the cache to make room for new data.

The goal of a caching system is to hide the high latency of the back-end database, by residing frequently used data closer to the computational resources, thus reducing the overall access time and improving system performance.

The two primary requests in caching APIs are: (1) **GET request:** Retrieving data object's value from the system given an object's key (2) **SET request:** Inserting data (i.e. key-value pairs) into the system. While there are additional request types, such as DELETE (delete data from the system), CAS (Compare-And-Swap), and others that may be specific to a particular cache design, it would be sufficient to focus on GET/SET requests from performance perspectives.

The below Figure 2.1 and Figure 2.2 illustrate the basic GET/SET workflow.

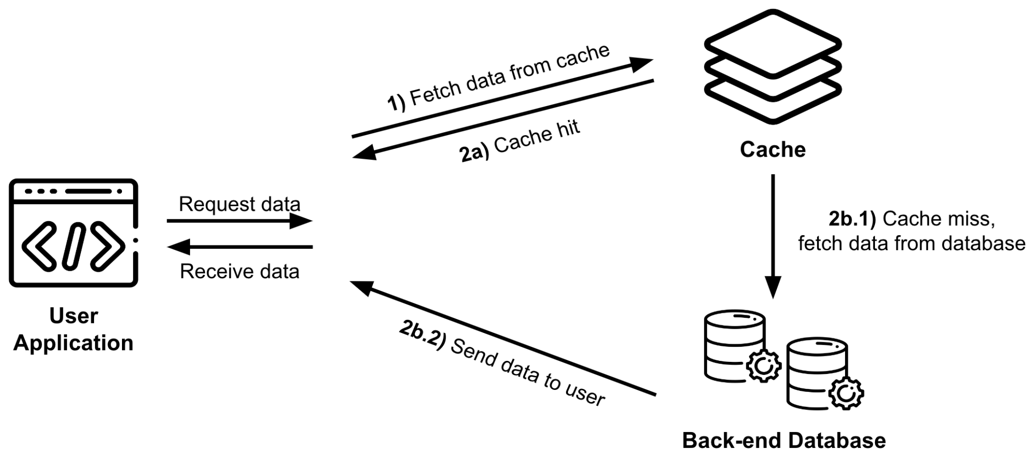

Figure 2.1: Basic GET Request Workflow

When a GET request is made for a data object, the system first checks if the object is available in the cache. If so, the data can be retrieved quickly without visiting the back-end store. We call such cases as *cache hits*. If not, then the system needs to retrieve the object from its back-end store. We call such cases as *cache misses*. A request that results in a cache miss usually takes a much longer time to receive its object, compared to the ones that result in a cache hit.

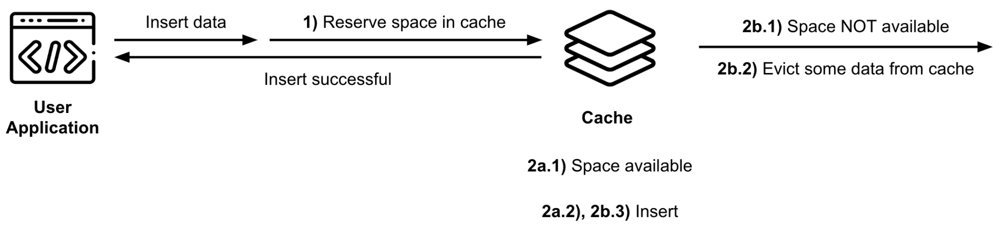

Figure 2.2: Basic SET Request Workflow

When a SET request is made to insert a data object, the system first checks if there is enough space in the cache to insert such an object. If so, the object can be inserted into the cache directly. If not, then the system needs to evict part of the data that was previously in the cache, where the decision to select these data is done by running the eviction algorithm; the evicted objects are then deleted from the cache, freeing up space so that the insertion can proceed.

Caching System Evaluation

Caching efficiency is measured in terms of *throughput* and *hit ratio*: (1) Throughput is the rate at which a cache can handle data retrieval and storage operations. A higher throughput indicates a more efficient caching system with better performance. (2) The hit ratio is the proportion of requests that result in a cache hit, out of all the GET requests, i.e., Hit Ratio = $\frac{\text{\# cache hits}}{\text{\# Total GETs}}$. A higher hit ratio indicates a more effective caching system with fewer cache misses, which means the overall access delay is small.

The Importance of Caching

The fundamental motivation of caching is to serve read/write requests faster. Most data queries exhibit a certain degree of data localities:

- *Temporal locality*: Data requested recently is likely to be requested again.
- *Spatial locality*: Data stored physically close to the data requested is likely to be requested soon.

Caching exploits such localities by storing the data objects that are likely to be accessed soon, hence reducing the access latency of these objects.

A well-designed cache can reduce the request traffic to the original data source, boosting the system to handle more requests. This can create an illusion of fast-large storage for applications, which leads to many benefits, including higher application performance, lower database cost, less back-end server load, more predictable performance, and network traffic, better user experience, less network congestion, less server workload, less energy usage [4], etc., – and ultimately enhance the overall user experience and increase company profitability [5, 6, 7].

Caching also serves an essential role in system robustness and reliability, as caching can reduce the load on individual components. Moreover, caching is essential for improving security and data protection. By storing frequently accessed data in a cache, the back-end store can be protected from malicious attacks such as Distributed Denial-of-Service (DDoS) attacks [8].

Other Types of Caching

There are different types of caching of different scales, such as Central Processing Unit (CPU) caching, Operating System (OS) caching, Content Distributed Network (CDN) caching, etc. Although they have different scales and usage, the underlying fundamental principles are similar.

2.2 Cache Eviction Policy

Cache, by its very nature, employs smaller and faster memory compared to the larger, slower back-end database storage. Due to its limited space capacity, when a cache becomes full, it must evict some of its current contents to make room for new data. *Caching eviction policy*, also known as *caching replacement policy*, is a method used to determine which objects to remove from a cache when it is full.

2.2.1 Policy Objective

The goal of a caching replacement policy is to remove the objects that are least likely to be used in the future, so that the most frequently used objects can be kept in the cache. The eviction process involves selecting and removing certain objects. An ideal eviction policy should minimize cache misses while keeping the computational overhead of the eviction process at a minimum.

We can formalize the objective of caching using mathematical notations. Suppose that:

$$\begin{aligned} h &: \text{cache hit ratio} \\ m &: \text{cache miss ratio} = 1 - h \\ T_h &: \text{data retrieval delay when there is a hit} \\ T_m &: \text{data retrieval delay when there is a miss} \\ H &: \text{caching system overhead} \end{aligned}$$

Then the objective of an eviction policy is to minimize the average data retrieval delay: $D = h \times T_h + m \times T_m + H$.

2.2.2 Heuristic Eviction Policies

Traditional cache eviction policies are heuristic-based, relying on certain predictable patterns of data use. These policies are simple, easy to implement, and have been widely used in various caching systems. The most common ones are:

- Least Recently Used (LRU): Evicts the object that has not been used for the longest period of time.
- Least Frequently Used (LFU): Evicts the object that has been used the least number of times.
- First In First Out (FIFO): Evicts the object that was added to the cache first.
- Random: Randomly evicts an object from the cache.

While these heuristic-based policies have proven effective in certain situations, they are based on certain assumptions or heuristics rather than an optimal analysis of the data access patterns. Therefore, heuristic-based policies may not perform well in scenarios where access patterns are complex or unpredictable.

2.2.3 More Advanced Eviction Policies

Caching eviction policies have undergone a significant transformation from traditional heuristic-based policies. Emerging strategies are characterized by their intelligent adaptability to varying application scenarios and traffic loads, in contrast to the heuristic-based policies that rely on a single metric.

Some state-of-the-art eviction policies are:

- Adaptive Replacement Cache (ARC): Dynamically adjusts the number of objects to be replaced based on the recent history of cache accesses [9].
- Cacheus: Supports multiple eviction policies, including LRU, LFU, and ARC; and dynamically adapts the policies based on the workload [10].
- Segmented LRU (SLRU): A variant of LRU that maintains two lists of objects: a probationary segment for new objects and a protected segment for objects that have been accessed more than once [11].

There are also many policies that incorporate external information to enhance caching performance. For instance, context-aware proactive caching [12] involves learning the context-specific content popularity of connected users and subsequently updating the cached content. Similarly, predictive caching policies have also been proposed to anticipate the popularity of multimedia content and proactively cache it [13, 14, 15].

Recently, caching policies have been improved by applying machine learning techniques, which have demonstrated promising outcomes in reducing response times and improving cache hit rates. A thorough discussion of current research in ML-based caching policies will be presented in Chapter 3.

2.3 Challenges in Data Center Caching

Designing and managing caches within data centers is a complex task. Despite the numerous advantages caching systems bring, they also introduce challenges that must be carefully addressed to ensure optimal system performance:

- Cache Misses:
  A cache miss occurs when requested data is not found in the cache, requiring the system to fetch it from the back-end disk, causing a delay ranging from milliseconds to a few seconds. Such high latency can result in slower application performance and low user satisfaction. In order to mitigate cache misses, it is crucial to employ effective strategies such as optimal eviction policies, prefetching, or increasing cache capacity.
- Scalability Constraints:
  With the ever-increasing data demand in today's digital world, scalability has become a crucial challenge. As the volume of data and the rate of data access

increases, it is necessary to design a caching system that can effectively scale up to handle the growing workload without significantly increasing latency or cost. In-memory caches, which only rely on DRAM, encounter scalability issues due to the limited space offered by DRAM.

- Hardware Limitations:
  Hardware limitations also present challenges in cache design. Balancing factors such as memory type, memory cost, cache size, access speed, cost, power consumption, and physical space can be complex.

These challenges demonstrate the complexity of cache design and management. Addressing these challenges effectively is critical to fully realize the advantages offered by caching systems in data centers. It sets the stage for our discussion on our HyperCache design, which attempts to address some of these challenges in innovative ways.

2.4 Memory and Storage Hardware

This section aims to elaborate on the key characteristics of DRAM and PMem, which serve as the fundamental memory technology used in HyperCache. We will highlight their respective physical characteristics, strengths and limitations, which justify their incorporation into a hybrid, high-capacity caching system. We will also discuss SSDs as a comparable alternative to PMem.

2.4.1 DRAM

Dynamic Random-Access Memory (DRAM) is a type of semiconductor memory that is widely used in modern computers as primary system memory.

DRAM stores each bit of data in a separate tiny capacitor within an integrated circuit. The capacitor can either be charged or discharged; these two states are interpreted as two different values of a bit, 0 or 1, respectively. Because the charge slowly leaks away, DRAM requires its data to be refreshed periodically to maintain the integrity of its data [16].

Strengths and Limitations

The primary strength of DRAM is its speed. It provides high-speed data access, typically in the nanosecond range, making it an excellent choice for applications requiring quick read/write operations. Its architecture also allows for random access, meaning any data byte can be accessed without touching the preceding bytes.

However, DRAM has its limitations. It is volatile memory, meaning it requires power to retain data. If the system is shut down or loses power, all data stored in DRAM is lost. DRAM also consumes more power compared to non-volatile memory, as it requires constant power and refreshing to maintain the data stored within.

Moreover, DRAM has a limited capacity compared to other memory technologies, such as hard disk drives and SSD. The smaller DRAM cell sizes become, the more susceptible they are to errors caused by electrical interference, charge leakage, and other physical effects that can cause data corruption and security vulnerabilities such as the “RowHammer Problem” [17]. Therefore, it is physically difficult to expand DRAM’s current space capacity. This capacity limitation can be a significant barrier for caches that require high-speed access to large datasets.

Lastly, DRAM, despite advances in manufacturing, still carries a relatively high cost per gigabyte. As of April 2023, the average cost of DDR5 DRAM[1] was around $USD 11.50 per gigabyte [18].

[1]DDR5 DRAM: Double Data Rate 5 DRAM is the latest (fifth) generation of Double Data Rate synchronous dynamic random-access memory

2.4.2 SSDs

Solid-State Drives (SSDs) are non-volatile, block-addressable storage devices that store persistent data on solid-state flash memory. There are several different types of SSDs available in the market, including SATA[2], SCSI[3], SAS[4], PCIe[5], and PCIe NVMe[6]. SSDs utilize NAND-based flash memory with data stored in a grid structure of cells, enabling fast data retrieval and modification [19].

Strengths and Limitaiton

One key strength of SSDs is their non-volatility, which ensures that data stored in SSDs is retained even during power disruptions, providing a level of data safety that DRAM cannot offer. Moreover, SSDs typically offer a lower cost per gigabyte than DRAM, making them a more affordable option for storing larger amounts of data.

However, SSDs do have limitations. SSDs have a finite number of write cycles, after which the cells can become unreliable. This phenomenon, known as wear-out, can potentially lead to data loss. Additionally, SSDs are significantly slower in read/write speeds compared to DRAM and PMem. This slower speed can impact performance in applications that require high-speed data access [19, 20].

2.4.3 PMem

Persistent Memory (PMem) is a high-capacity, non-volatile memory technology. There are various kinds of PMem, including Intel's 3D XPoint technology used in Optane DC Persistent Memory modules, which can offer memory capacities much higher than conventional DRAM [21].

Strengths and Limitations

Similar to SSDs, one major strength of PMem is that it can retain data even when power is off. PMem also has a high space capacity, allowing for the storage of larger datasets closer to the processor, which can speed up data access times.

PMem offers the additional benefit of being byte-addressable, similar to DRAM. This attribute enables PMem to function more like memory rather than storage. SSDs, on the other hand, are block-addressable, which means they must be read or written in larger chunks.

However, despite being faster than traditional storage options like SSDs, PMem is generally slower than DRAM in terms of access speed [1].

[2]SATA: Serial AT Attachment
[3]SCSI: Small Computer System Interface
[4]SAS: Serial Attached SCSI
[5]PCIe: Peripheral Component Interconnect Express
[6]NVMe: Non-Volatile Memory Express

Table 2.1 provides a summary of the comparison among these three types of memory technologies.

Characteristic	**DRAM**	**SSD**	**PMem**
Speed	High	Medium	High (lower than DRAM)
Volatility	Volatile	Non-volatile	Non-volatile
Capacity	Low	High	High
Cost per GB	High	Medium	High (lower than DRAM)
Endurance	High	Low	Low
Byte-addressability	Yes	No	Yes
Power Consumption	High	Low	Medium

Table 2.1: Comparison of DRAM, SSD, and PMem

In conclusion, PMem represents a middle ground between DRAM and SSDs and integrates the strengths of both: the byte-addressability and near-DRAM speeds of DRAM; and the non-volatility and high capacity of SSDs. Such characteristics make PMem a promising solution for handling large amounts of data rapidly and persistently. As a result, our HyperCache has been designed to use DRAM and PMem in its first prototype.

2.5 Segcache

Pelikan [22] is a framework engineered by Twitter. It is designed to be fast, reliable, and modular, providing high-throughput and low-latency caching solutions. Pelikan is integral to Twitter's infrastructure, functioning as its production caching system.

Within the Pelikan framework, Segcache [2] stands out as a cutting-edge in-memory cache specifically designed for the efficient handling of small objects. This makes Segcache optimal for web applications demanding high-throughput and low-latency services. Our work HyperCache is designed and implemented upon Segcache.

In this section, we provide an overview of Segcache, highlighting its distinctive architecture, features, and benefits.

2.5.1 Segcache Key Motivations

The development of Segcache was primarily motivated by three critical issues identified in existing in-memory caches, which significantly hindered their performance and memory efficiency:

- Large metadata overhead
  Modern web applications commonly handle small objects ranging from tens to thousands of bytes [23, 24]. However, traditional in-memory caches are not designed to handle such small objects efficiently, resulting in large per-object metadata. These metadata consumes a substantial portion of the memory, thereby reducing the overall memory available for actual data storage.
- Delayed expiration
  In existing caching systems, expired objects are not promptly removed, leading to these objects occupying valuable memory space.
- Slow expiration speed
  In existing caching systems, the process of identifying and removing expired objects is often slow and resource-intensive, thereby reducing the overall throughput of the caching system.

2.5.2 Segcache Architecture and Design

Segcache's architecture is designed to maximize memory efficiency, throughput, and scalability. It consists of three main components: (1) A TTL-indexed bucket array; (2) An object-store made up of segments; (3) A hash table for object lookup.

TTL Buckets

Segcache organizes data objects into separate ranges known as TTL (Time-To-Live) buckets, with each bucket having its specific range. Within each TTL range, every object is assigned the lower bound of the range as its TTL. This design not only reduces the metadata overhead associated with individual object TTLs, but also facilitates proactive expiration of objects.

Object Store: Segments

In Segcache, objects (i.e. key-value pairs) are organized and managed in segments. Segments are the primary data management units. Segcache groups objects based on their approximate TTL and creation time.

Segments are associated with their corresponding TTL. Each TTL bucket stores pointers to the head and tail of its time-sorted segment chain, with the head segment being the oldest. Objects are sequentially appended to a segment's end, ensuring a chronological ordering of segments and their internal objects. Segments are linked into chains.

During expiration, Segcache identifies the first expired segment within each TTL bucket, subsequently considering all parent segments in the same TTL linked list as expired, since those segments must have an earlier creation time by construction. This streamlined design enables a collective expiration process, which is highly efficient and straightforward.

Hash Table

Segcache uses a bulk-chaining hash table similar to other systems like MICA [25] and Faster [26]. objects that share the same hash are grouped into a hash bucket. Each slot within the bucket records the object's key tag, the segment it resides in, and its offset within that segment. The key tag associated with each object plays a crucial role in reducing hash collisions: Upon object look-up, the query key is hashed into a tag and compared with the hash table tags. When the tag does not match the provided key, it becomes unnecessary to inspect the segment and compare the actual object key.

The novelty is that each hash bucket shares common metadata of objects within the same bucket, such as access timestamp and read/write spin-lock. This "shared info slot" is allocated at the beginning of each hash bucket for convenience of access. Such hash bucket design significantly reduces metadata overhead, contributing to the overall memory efficiency of Segcache.

The below Figure 2.3 from the original Segcache paper [2] intuitively illustrates its architectural design:

A **read** request starts from the hash table on the right. The key provided by the request is used to look up the corresponding bucket in the hash table. If the bucket contains a slot that has a matching tag with the provided key, then Segcache retrieves the object from the corresponding segment. A **write** request starts from the TTL buckets on the left. The TTL of the object is used to find the corresponding bucket. The object is then appended to the segment at the tail of the segment chain associated with the bucket. If the tail segment is full, a new segment is allocated and added to the bucket. The key and the segment are then inserted into the hash table.

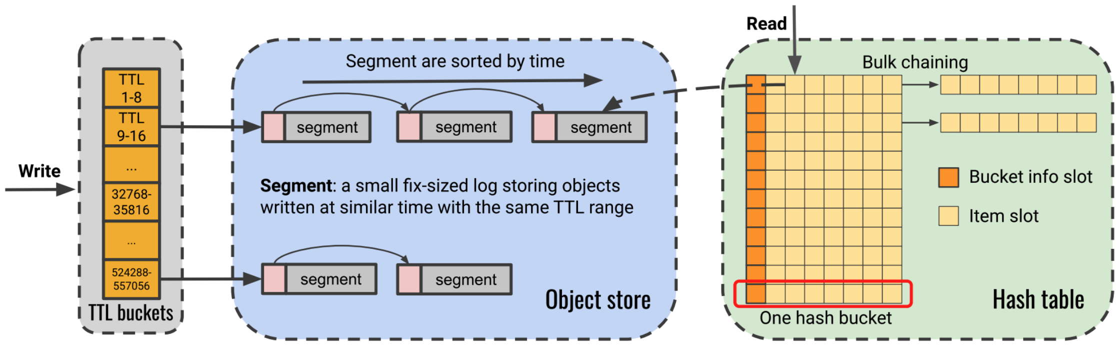

Figure 2.3: Overview of Segcache [2]

We also provide pseudo-code algorithms for Segcache read, write, and expiration operations in Appendix A block 1, 2 and 3.

2.5.3 Benefits of Segcache

Segcache's novel design significantly reduces per-object metadata overhead, leading to superior memory utilization. This is especially beneficial for caching small data objects, a common requirement in modern web applications. Furthermore, Segcache's segment-based macro management enhances scalability by minimizing locking overhead, thereby enabling high throughput even under heavy loads.

The segment-based macro management offered by Segcache is a powerful feature that can be utilized in the design of a hybrid DRAM-PMem cache. This approach enables efficient data placement and movement between volatile DRAM and non-volatile PMem by operating at the segment level rather than at the individual object level.

2.6 Machine Learning

Machine learning (ML) is a type of artificial intelligence that allows computers to learn without being explicitly programmed. Machine learning algorithms are trained on data, and they use that data to make predictions or decisions. This ability makes ML a potent tool in various fields, including data center management and optimization.

In this section, we explore the ML algorithms used to optimize cache eviction decisions in our HyperCache design.

2.6.1 Supervised Learning

Supervised machine learning is a branch of artificial intelligence that involves training a model on labeled input-output pairs, by making predictions or decisions by generalizing from the provided examples. A model is usually trained using algorithms that minimize the discrepancy between its predictions and the true output values. Once trained, the model can make predictions on new, unseen data based on the learned patterns from the training data.

This methodology aligns with our goal of predicting future data object utilization. Therefore, we employ the following machine learning algorithms in the eviction mechanism of our HyperCache design.

Linear Regression

Linear Regression (LR) is a simple supervised machine learning algorithm that can be used to predict a continuous value from a set of features. It works by fitting a straight line to observed data. The line is determined by minimizing the sum of the squared differences between the predicted values and the actual values.

The advantages of linear regression include simplicity, ease of interpretation, and computational efficiency. However, it assumes a linear relationship between the variables and may not perform well with complex or non-linear data patterns.

Decision Tree

Decision Tree is another type of supervised machine learning algorithm that works by organizing data into hierarchical tree structures, hence the name “decision tree”. Each internal node represents a feature, and each leaf node corresponds to a predicted outcome or class label.

Decision trees are easy to interpret and can capture non-linear relationships. However, they are prone to over fitting, can be sensitive to small variations in the data, and may not generalize well to unseen examples.

In our work, we use XGBoost (eXtreme Gradient Boosting) [27] which is built on top of decision trees.

Neural Network

Neural networks (NN) are more complex machine learning algorithms that can be used to predict both continuous and categorical values. They are designed to mimic the human brain's functioning by using layers of artificial neurons (or nodes). These networks are capable of learning complex patterns and relationships from large volumes of data, making them highly applicable to diverse tasks.

However, they require substantial computational resources, and extensive training data, and can be challenging to interpret compared to simpler models like linear regression or decision trees.

Algorithm	**Advantages**	**Disadvantages**
Linear Regression	Simple and easily interpreted	Assumes linear relationship
Decision Tree	Easy to interpret	Prone to over fitting
Neural Network	Learns complex patterns and relationships	Requires more computational resources

Table 2.2: Comparison of Linear Regression, Decision Tree, and Neural Network

Table 2.2 provides a brief comparison of these algorithms.

2.6.2 Measuring the Significance of Features

In machine learning, it is important to understand whether a certain feature is related to the target result. Features are the variables that are used to train an ML model. Effective pre-analysis and feature selection not only contribute to reducing model complexity, but also enhance model accuracy by eliminating interference from irrelevant features.

In this section, we will discuss Correlation Analysis and Random Forest Importance, which are the two methods we used in optimizing ML eviction for HyperCache.

Correlation Analysis

Correlation analysis is a statistical method used to assess the magnitude and direction of the linear association between two variables. The correlation coefficient, which varies between -1 and 1, quantifies this relationship correlation. A correlation coefficient of +1 indicates a perfect positive correlation: it means that as one variable increases, the other variable also increases. A correlation coefficient of -1 indicates a perfect negative correlation: it means that as one variable increases, the other variable decreases. A correlation coefficient of 0 means no correlation between the two variables.

Random Forest Importance

Random forest is a machine learning algorithm that can be used for both classification and regression tasks. Random forest works by constructing a large number of decision trees and then combining the predictions of the trees to make a final prediction [28].

Random Forest Importance, also known as “Gini importance” or “Mean Decrease in Impurity,” is a technique employed in ML models based on the Random Forest algorithm. It accesses the importance of a feature by counting the number of times this feature is used to split a node in the random forest tree. The importance of each feature is then normalized to a range of 0 to 1. Features with higher importance scores have a more substantial impact on the model’s output [29].

Therefore both Correlation Analysis and Random Forest Importance can help in feature selection, identifying relevant variables, and improving model interpretability by highlighting the most influential features.

Chapter 3

Related Work

In this chapter, we overview the most popular in-production high-capacity data center caches, as well as current research on hybrid caches to expand capacity beyond DRAM. Then, we look into prior studies examining the utilization of PMem in other applications, such as transaction management, memory management, and file systems. Lastly, we review recently proposed machine learning techniques for cache evictions, which seek to further improve cache effectiveness.

3.1 In-Production Data Center Caches

Caches are a critical component of modern data-centric software systems. Companies like Twitter, Facebook, and Amazon heavily rely on such caches to reduce data access latency and enhance the performance of their services. This section provides a comprehensive comparison of five leading cache systems: Pelikan, Cachelib, Redis, Memcached, and Amazon ElastiCache. Each of these systems offers unique features and capabilities that cater to specific requirements.

Pelikan [22] is a highly efficient, open-source cache system that stands out for its performance and flexibility. It utilizes a modular architecture that allows for easy customization and integration with existing infrastructure. This cache system is designed to provide low-latency access and high throughput, making it particularly suitable for demanding applications with complex caching requirements.

Cachelib [30], on the other hand, focuses on optimizing cache efficiency by leveraging various techniques such as probabilistic filters and compressed representations. This cache system prioritizes memory utilization and cache hit rates, making it an excellent choice for scenarios where memory capacity is limited. Cachelib's emphasis on efficient resource usage makes it suitable for applications that require high cache utilization and a low memory footprint.

Amazon ElastiCache [31], a fully managed caching service provided by Amazon Web Services (AWS), offers the convenience of cloud-based caching. ElastiCache supports both Redis and Memcached. It provides automatic scaling, replication, and backup capabilities, along with seamless integration with other AWS services. ElastiCache simplifies cache management and deployment, making it an attractive option for organizations leveraging AWS infrastructure.

Redis [32] and *Memcached* [33] are generalized cache libraries that offer a wide range of functionalities for caching, including real-time analytics, session management, and caching in web applications. Its support for clustering and replication also ensures high availability and fault tolerance. Redis and Memcached are well-established caching systems with long histories[1] and widely adopted APIs (Application Programming Interfaces).

The below Table 3.1 provides a brief comparison across current in-production caching systems of data centers.

Cache	**Developed By**	**Used By**
Pelikan	Twitter	Twitter
CacheLib	Facebook	Facebook
ElastiCache	AWS	HBO Max, Yahoo, Airbnb
Redis	Redis	GitHub, StackOverflow, Snapchat
Memcached	Danga Interactive	Youtube, Wikipedia, Reddit

Table 3.1: Current In-Production Data Center Caches

[1]Redis was created in 2009 and Memcached was created in 2003

3.2 Hybrid Caching

The increasing demand for data-intensive applications, coupled with the cost and space constraints associated with DRAM, has prompted a rising interest in hybrid cache systems. These systems integrate two or more memory types, such as DRAM and SSDs, to achieve a balance between cache space capacity, performance and cost.

Several works provide valuable insights into this topic. One notable instance is Facebook's RocksDB [34], an embedded hybrid persistent key-value store. However, despite its innovative design, RocksDB does not adequately account for SSDs, and it presents higher write and space amplification compared to alternative storage engines.

Further building on this foundation, Facebook introduced “MyRocks” [35], a system aimed at reducing the DRAM footprint in data centers through the incorporation of Non-Volatile Memory (NVM). MyRocks is constructed atop the RocksDB storage engine, resulting in lower DRAM and storage space requirements compared to InnoDB, which is the default storage engine. However, the researchers of “MyRocks” also noted that MyRocks is not universally applicable [35].

Another work, “Kangaroo” [36], was proposed in 2021 to address SSDs' write amplification issue in hybrid caches. Kangaroo combines a large, set-associative cache with a small, log-structured cache to reduce both DRAM usage and flash writes. It is particularly optimized for small objects that are 100 bytes or less. While not yet in production use, Kangaroo is considered the state-of-the-art for caching tiny objects in large-scale hybrid caches.

Current Gaps

While existing studies offer valuable insights, hybrid researches and designs often lag behind in-memory designs due to their inherent complexity. To date, no hybrid caching system has employed a TTL indexing mechanism as incorporated in the Segcache design [2]. This represents a gap in the literature that our research intends to address.

Furthermore, given that PMem is a relatively new memory technology, the majority of studies have focused on SSDs hybrid caches instead. These designs have been optimized for SSDs-specific characteristics such as their block-addressing feature [37]. Therefore, a hybrid cache integrating DRAM and PMem could offer meaningful insights, potentially heralding a new research field in data center caching.

3.3 PMem Applications

PMem (Persistent Memory) is a new memory technology [21] developed in recent years that is making a significant impact on the field of computing. It has distinct characteristics combining the advantages of both primary and secondary storage. PMem is byte-addressable, offering comparable speed to DRAM, while also providing ample capacity, persistence, and cost-effectiveness similar to secondary storage. A number of recent studies have proposed software stacks for PMem applications, including transaction management, memory management, file system management, and whole system persistence. These studies provide a valuable foundation for future research on PMem.

3.3.1 Transaction Management

Transaction management is a set of processes and techniques that are used to ensure the consistency and integrity of data in a database. This study [38] presents an approach that leverages the byte-addressable nature of PMem to attain high-performance transactions. Their study not only sets a benchmark for transaction performance in PMem applications but also demonstrates the potential benefits of PMem's unique properties in this context. Similar to this research, "Blurred Persistence" is another frame work that aims at further enhancing system efficiency through the utilization of PMem [39].

3.3.2 Memory Management

Regarding memory management systems that employ PMem, [40] proposes a novel approach aimed at ensuring consistency, durability, and safety using PMem. On the other hand, [41] explores the use of PMem for achieving crash consistency in persistent memory systems, emphasizing the advantage of PMem's volatility. The work "Whole System Persistent" [42] further presents a more ambitious approach that utilizes PMem for the entire system memory.

3.3.3 File System

In terms of PMem-based file systems, Ziggurat [43], introduces a tiered file system that combines Non-Volatile Main Memory (NVMM) and slow disks to create a storage system with near-NVMM performance and large capacity. Ziggurat optimizes disk bandwidth utilization by leveraging data access patterns, write size, and application stall likelihood to migrate cold file data from NVMM to disks. Meanwhile, Strata [44] introduces an innovative file system that leverages the advantages of various storage media, including PMem, SSD, and HDD. Strata employs a log-structured method that distributes responsibilities across user mode, kernel, and storage layers uniquely, achieving high-performance storage layer management.

In summary, these studies demonstrate the benefits of PMem in terms of performance, consistency, and durability. They offer us insightful understandings of the effective utilization of PMem within a wide range of applications.

3.4 Machine Learning in Cache Eviction

Traditionally, cache eviction has been handled using heuristics. These heuristics are based on simple rules of thumb, such as the First-In-First-Out (FIFO) and the Least-Recently-Used (LRU) policies. However, as discussed in Chapter 2, Section 2.2, heuristic-based eviction policies can be sub-optimal, especially in dynamic workloads.

In recent years, the application of machine learning (ML) techniques to cache eviction strategies has emerged as a promising area of research. This section discusses three main categories of studies within this field, including object-level learning, learning from simple policies, and optimization studies to improve the efficiency of ML in cache evictions.

3.4.1 Object-Level Learning

Object-level learning takes a granular approach to cache eviction by making decisions based on individual data objects or cache lines. Many studies have been made to predict data objects' reuse distance and popularity [3, 45, 46, 47], and evict objects that are less likely to be accessed again.

A representative work in this field is the "Learning Relaxed Belady Algorithm" (LRB), introduced in [3]. LRB is a novel cache eviction policy that incorporates machine learning into the classic Belady Algorithm [48]. LRB utilizes a Gradient Boost Machine [49] to predict objects' future access time. It evicts the first object it finds with a future access time beyond a specified threshold. Compared to the classical Belady algorithm, LRB reduces computational overhead by relaxing the constraints.

However, despite their relatively high accuracy, these methods are complex and computationally expensive. Not only do they require a substantial computational overhead for training and predicting, but they also require significant storage for collecting training data, which occupies the valuable space of the in-memory cache.

3.4.2 Learning from Simple Policies

Another field of ML in cache eviction involves the use of reinforcement learning (RL) [50] to adapt decisions among simple policies such as FIFO, LRU, and LFU [10, 51].

For instance, Cacheus [10] employs a collection of experts (i.e. simple eviction policies), consisting of LFU, ARC, LIRS, SR-LRU (a scan-resistant version of LRU), and CR-LFU (a churn-resistant version of LFU) [9, 52]. Cacheus uses RL to choose among these experts and updates experts' weights based on their past decisions and performance.

However, this approach may not be flexible enough to adapt to dynamic workloads quickly. The RL model can be impacted by the momentum of previous workloads. Also, the weight-adjusting process may not be prompt enough. Additionally, RL models can be complex and computationally expensive.

3.4.3 Optimisation ML in Cache Eviction

The computational overhead has been a major challenge in applying ML in cache eviction. This has led to interest in studies aimed at improving the efficiency of ML techniques for cache eviction. In this section, we will look at two notable efforts in this direction: (1) Machine Learning At the Tail (MAT) [53]; (2) Group-Level Learning (GL-Cache) [54].

MAT aims at reducing the computational overhead in ML caching by using a hierarchical framework. MAT uses traditional heuristic policies as “filters” to procure high-quality samples for ML training, significantly reducing the complexity of ML training and predictions. This work contributes to the field by offering a practical and efficient ML approach that can be integrated with existing heuristic-based systems. However, this approach is sensitive to the filter selection. There are instances where other algorithms (e.g. LRB) demonstrate slightly better performance than MAT [53].

GL-Cache is a unique ML-based cache eviction design that employs a different approach to predicting future access likelihood. Instead of focusing on individual objects, it operates at the level of groups. This innovative strategy brings several benefits, including a reduction in the overhead associated with collecting training data, the ML training process, and the complexity of the ML model.

However, despite its promising framework, GL-Cache still relies on individual object selections after identifying the target group to achieve a high hit ratio. This makes the eviction process lengthy and does not fully exploit the potential advantages of group-level eviction. As a result, in systems that utilize PMem or SSDs, the performance of GL-Cache might suffer due to the high latency associated with these memory technologies.

Current Gaps

In the above work, ML-based cache eviction techniques have demonstrated promising results in improving hit ratios and outperforming traditional heuristic-based policies. However, these approaches have limitations, including overheads in data collection and training processes, impacting memory efficiency and cache throughput.

While previous research has explored efficiency improvements in cache eviction, such as GL-Cache and MAT, this research field remains largely under-explored. The majority of the existing work in this field primarily emphasizes object-level learning to effectively utilize the information present in the current cache. However, these approaches may not be well-suited for scenarios involving a second-level cache. In our specific case, the costs associated with first-level cache misses are relatively low due to the availability of evicted objects in PMem. As a result, the current approaches may not be applicable or suitable for our HyperCache design.

Chapter 4

Experimental Methodology

In this chapter, we discuss our experimental methodology, setup, the data-center request traces we use for experimentation, and the evaluation metrics. We discuss methodology first as key quantitative findings inform the design of our newly proposed caching approach.

4.0.1 Experiment Traces

The experiment was conducted using a range of real-world traces recorded from different sources. This ensures that the experiments mirror realistic workloads and that the experiment's requests were predictable and unaffected by external factors. The two main datasets used are: (1) 14 traces from Microsoft Research Cambridge (MSR)[1] [55] and (2) 53 traces from CloudPhysics[2] [56].

4.0.2 Evaluation Platform

All the experiments were conducted on a Dell R740 server with the following specifications in Table 4.1:

This hardware setup provided the necessary computational power and storage capacity for implementing and testing our HyperCache system and conducting all related experiments.

The experiments were conducted using the following methodology:

- Each individual trace was executed three times, and the subsequent results were averaged to ensure accuracy and reliability.

[1]MSR traces download link:
https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/fast23_glcache/msr/

[2]CloudPhysics traces download link:
https://ftp.pdl.cmu.edu/pub/datasets/twemcacheWorkload/fast23_glcache/cphy/

System	
Operating System	Ubuntu 18.04.1 Linux OS (5.4.0 kernel)
Hardware	Dell PowerEdge R740 Server
Processor	
Processors	Intel Xeon Gold 6252N
Number of cores	48 physical cores (96 logical)
Core frequency	2.3 GHz
Last-Level Cache	
L3 cache	shared 36 MB (one socket)
DRAM	
Capacity	384 GB
NVM	
Capacity	1.5 TB
Hardware	Intel Optane Persistent Memory
SSD	
Capacity	1 TB
Hardware	3.5-Inch, Seagate, SATA (6 Gbps)

Table 4.1: System parameters

- To avoid potential issues from overheating, a minimum gap of 30 seconds was deliberately maintained between each trace run.
- For data analysis, we used a default 1 GB cache size for the MSR datasets and a 2 GB cache size for the CloudPhysics dataset.
- For performance evaluations, we varied the cache size as later specified.

4.0.3 Evaluation Metrics

In our experiments, we employed two primary metrics to evaluate the performance of the HyperCache design: hit ratio and throughput, which were mentioned in Chapter 2.1.1.

In the context of this study, it is important to note that we discarded the long latency of cache misses in our throughput measurement. Hence the throughput is equivalent to the throughput of cache hits. The throughput unit we use is MQPS (Million Requests Per Second).

There are two rationales behind this: (1) In our experimental setup, we do not have a remote database server from which to retrieve data where there is a cache miss. (2) By focusing primarily on the throughput of cache hits, we obtain a throughput measure that is largely indicative of the cache's own performance, independent of network or database latencies that might be encountered when fetching data from a remote source upon a cache miss.

Chapter 5

Design and Implementation

In this chapter, we explore the possibility of developing a hybrid cache that utilizes DRAM and PMem. The goal of the system is to take advantage of both DRAM and PMem, as well as to combine the most efficient eviction algorithm. We discuss the design principles that guided our work, the architectural framework of HyperCache, and the optimization techniques in HyperCache's eviction policy which involves machine learning and bulk eviction.

The rationale of our design is supported by experimental analysis and observations. In this chapter, we also highlight the intermediate empirical results that motivate and support our design and optimization, providing key insights into our development process.

5.1 Design Principles

The conception and design of HyperCache are guided by a set of core principles. We aim at addressing the challenges in current data center applications which we pointed out in Chapter 2, Section 2.3. The primary principles shaping the design of HyperCache are as follows:

- **Efficient Usage of DRAM**
  Given that DRAM has a lower capacity and higher speed than PMem, its usage should be optimized for maximum efficiency. Thus the allocation in DRAM must be primarily focusing on data objects that require frequent access.
- **Minimize PMem Footprint**
  HyperCache can potentially have a larger capacity by using PMem. However, PMem has higher operation latency than DRAM. Therefore, HyperCache needs to only visit PMem when necessary, efficiently leveraging its large capacity while minimizing potential latency issues.

- **Simplicity and Maintainability**
  While our design is complex, we aim for high maintainability in both the design and implementation. This means that the system should be easy to configure, manage, maintain, and further develop.

5.2 HyperCache Architecture

In a hybrid cache, DRAM and PMem are combined in a tiered architecture, where DRAM is used as a cache for PMem. The goal is to keep the frequently accessed data (i.e. "hot" data) in DRAM, while less frequently accessed data (i.e. "cold" data) is stored in PMem.

Our HyperCache design builds upon the framework of Segcache's efficient TTL indexing and segment-structured design [2]. We integrate two linked lists of segments for each TTL bucket: one in DRAM and one in PMem.

Data allocation first goes to DRAM. If DRAM becomes full, we evict the data in DRAM into PMem using a DRAM-eviction policy. When PMem is full, we evict the data in PMem using a PMem-eviction policy. This policy can be different from the DRAM-eviction policy and can be designed considering the characteristics of PMem.

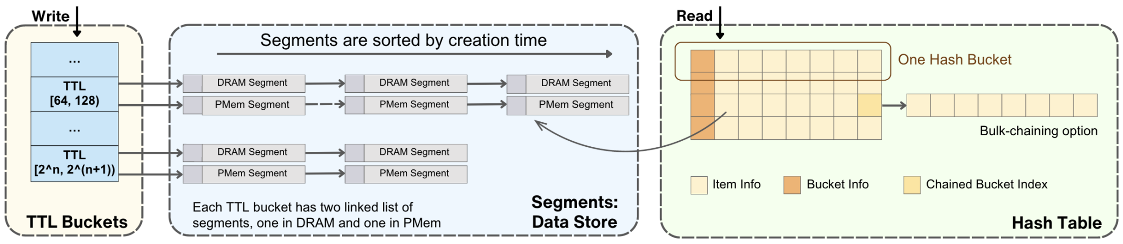

Figure 5.1: HyperCache Design Overview

A design overview is illustrated in Figure 5.1. The primary distinction between HyperCache and Segcache operations lies in the process of data writing. In Appendix A block 4, we present a pseudo-code algorithm for HyperCache write operations.

This design allows us to make full use of the innovative and efficient design of Segcache's TTL indexing, where objects are grouped into segments. This grouping facilitates efficient macro-management, reducing the overhead of managing individual objects and improving overall system performance.

5.3 HyperCache Eviction

5.3.1 Machine Learning Eviction

In this section, we will discuss the specifics of how we integrated machine learning into HyperCache eviction process. This includes integrating ML model, identifying features that are influencing eviction decisions, and ML algorithm selection.

Feature Selection

The first step in our eviction design is a systematic analysis of segment features, that can be potentially used in training and predicting future segment visit numbers for eviction. This design follows a collect-snapshot-collect paradigm, which is a similar technique as used in LRB Algorithm [3] and GL-Cache [54]. Figure 5.2 provides an illustration of the data collection workflow.

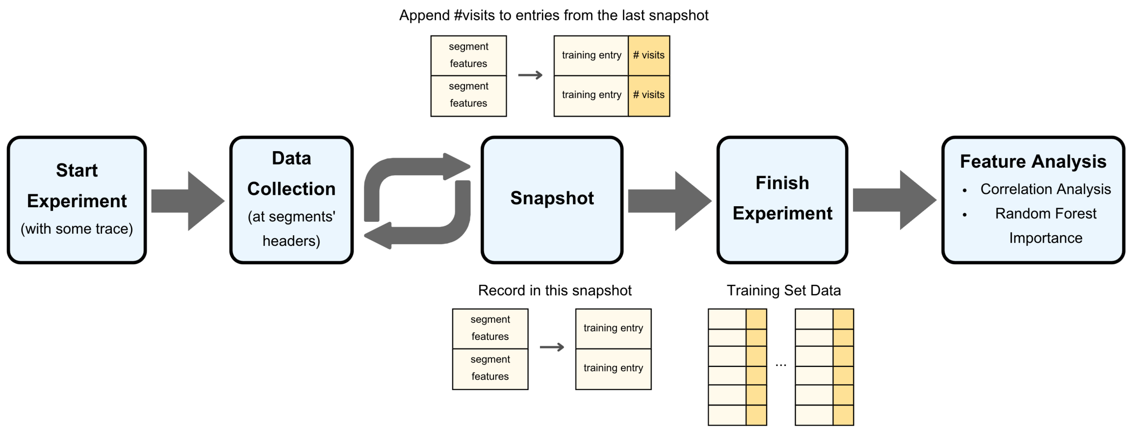

Figure 5.2: Feature Data Collection Workflow

As shown in Figure 5.2, the steps are as follows:

1. Data Collection:
   The first step is to record a feature set that could potentially have an impact on segment visits: read rate, write rate, number of previous misses, number of live objects, number of live bytes, age, creation time, the previous number of reads, and active objects (i.e. objects that have been accessed more than once).
   We allocated space in the segment header to record these features. Subsequently, these features were recorded during cache operations.

2. Snapshot Features:
   This step generates a snapshot that captures the current state of segment features at a fixed interval. During each snapshot, all features of every segment were recorded and added to the training set, with each segment having its own feature

entry. After capturing the snapshot, the features in segment headers were continually updated to prepare for future snapshots.
In our experiments, we use a 1-minute interval as our snapshot gap.

3. Get "Future" Number of Segment Visits:
Next, we gathered the number of visits to each segment in the following snapshot and linked them to their corresponding feature entries. This creates the feature-target training sets.
An ideal prediction algorithm would be able to predict the number of segment visits, which is the "target", from the segment features. Combined with the eviction algorithm, this prediction would allow the cache to accurately select the least useful segment for eviction.

4. Offline Analysis:
We collected the training set by collecting snapshots at regular intervals throughout the entire experimental trace. We then performed the feature importance analysis, as detailed in Chapter 2.6.2, to assess the significance of the features.

The experimental results for the MSR dataset and CloudPhysics dataset are shown in below Figure 5.3 and Figure 5.4.

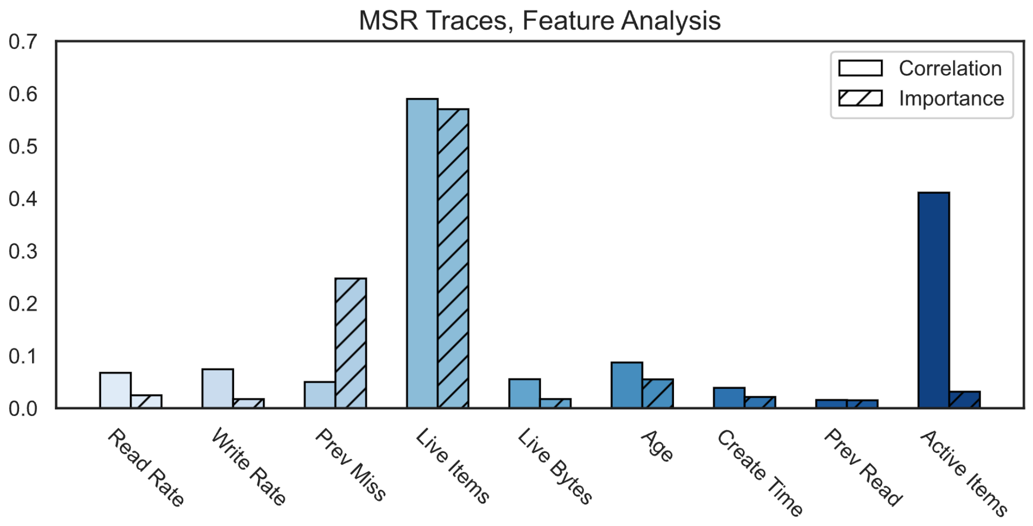

Figure 5.3: MSR Feature Analysis

The feature analysis of the MSR and CloudPhysics datasets revealed that the **live objects numbers** and the **active objects numbers** are the most important features. These features have the highest correlation and importance, indicating their substantial role in the model. **Age** and **previous misses numbers** were also relatively important features.

Although the correlation between the features and their importance is not always con-

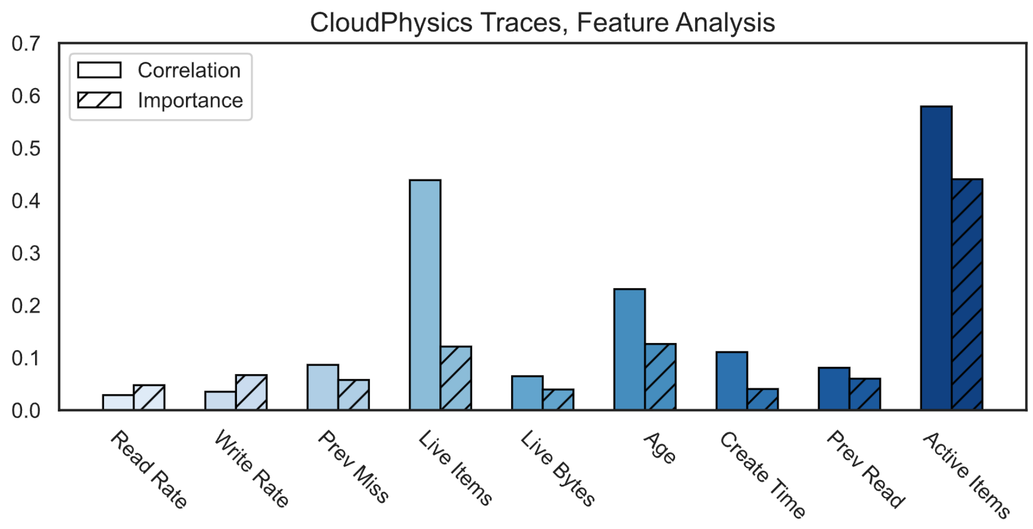

Figure 5.4: CloudPhysics Feature Analysis

sistent, we can conclude that other features in both datasets are less important. Considering that the collection process of these features also constitutes a large part of cache operation overhead, so we decided not to include these features in the feature collection and ML model training.

In summary, based on our feature analysis, we have chosen to include four key features in the eviction model. These include the number of live objects, segment age, number of active objects, and previous misses in the eviction decision process.

ML Model Selection

In this section, we compare three ML models: (1) Linear regression; (2) XGBoost; and (3) A small neural networks of two layers and 10 nodes in total. We aim to observe the trade-offs associated with different ML models for eviction policy selection. The objective of this set of experiments is to determine the most efficient and effective ML model for eviction policy in HyperCache.

Additionally, we used a set of heuristic-based eviction policies as baselines for comparisons. They are: Util (Utility: The number of accumulated visits to the Segment), FIFO (First-In-First-Out), CTE (Closest-To-Expiry).

The workflow of ML eviction follows a similar paradigm as described in Figure 5.2, with an additional training-prediction step. We inserted an ML training process after each snapshot, using the current valid feature-target entries as the training set. We also discard data from earlier snapshots, as it no longer represents the current status of cache workload traffic. This allows the model to dynamically learn the workload pattern and make prompt adjustments, such that the eviction decision is accurate and dynamically adapts to the cache workload traffic. After training, we rank the segments by performing predictions using their current features. The ranking is then used in eviction to determine the segment to evict.

To account for the distinct physical characteristics of DRAM and PMem, we performed a separate comparison of ML models for eviction in each memory type. This analysis aimed to capture any differences in the performance of the models across memory technologies.

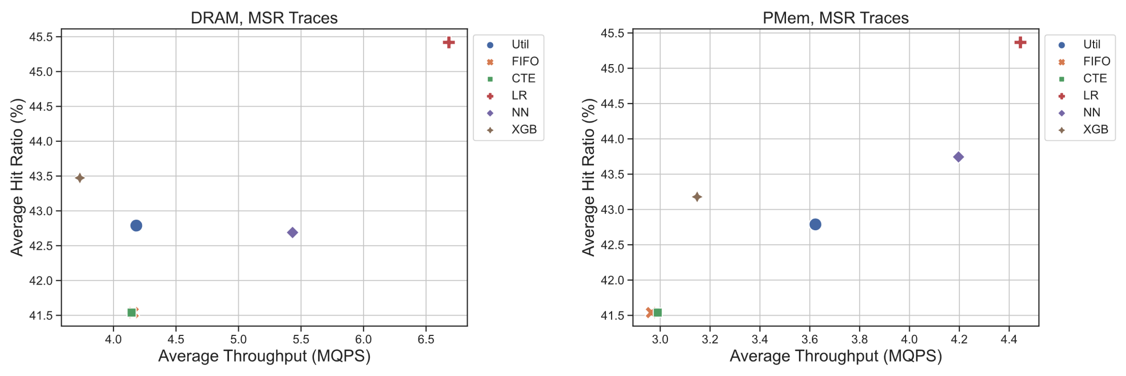

Figure 5.5: MSR Eviction Policies Comparison

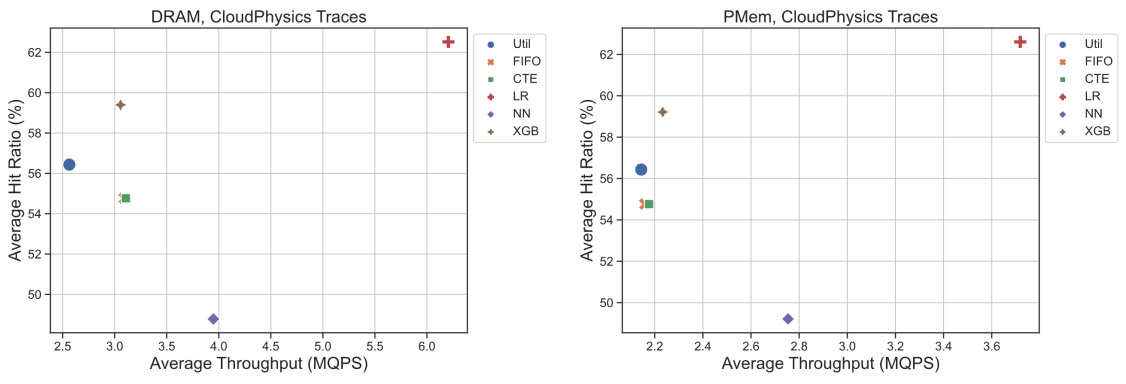

Figure 5.6: CloudPhysics Eviction Policies Comparison

As shown in Figure 5.5 and Figure 5.6, our experimental findings indicate that ML-based eviction policies outperform heuristic-based eviction policies in terms of cache hit ratio, demonstrating their better effectiveness in cache eviction. However, it is observed that ML models that use XGBoost and Neural Network exhibit lower throughput compared to heuristic-based models and the linear regression model. This outcome aligns with expectations due to the higher complexity and training overhead associated with these ML models.

The results show that linear regression (LR) is the most efficient and effective ML eviction policy. This result also aligns with our expectations: We carefully selected features that exhibit a strong correlation with the prediction of future segment visits. Consequently, the LR model performs well in making accurate predictions, as the selected features reveal clear patterns. This highlights the importance of feature selection, as it enables us to capture meaningful relationships between the input variables and the predicted outcome. This makes the LR-based eviction policy well-suited for our HyperCache design.

Another insightful observation is that the LR model demonstrates a higher throughput compared to the baseline heuristic-based eviction policies. This improvement can be attributed to the fact that heuristic-based policies require segment re-ranking upon eviction, whereas LR only performs re-ranking during snapshot windows. The ML prediction process of LR exhibits high accuracy, allowing for less frequent re-ranking. Furthermore, LR's simplicity results in minimal overhead, contributing to its higher throughput.

It may seem counter-intuitive that LR performs the best among the advanced ML models. However, it is important to note that the application of ML in low-level computer systems requires additional considerations, as in this context high-speed processing is a priority. This is in contrast to tasks such as natural language processing or computer vision tasks, which may not be as time-critical. Our work delivers meaningful insights into this aspect.

5.3.2 Optimization: Bulk Eviction

After selecting Linear Regression as the ML eviction model, we moved on to another optimization of the eviction policy: bulk eviction. Since our eviction operates in the unit of segments, we conducted a series of experiments to assess different bulk eviction sizes, ranging from 1 to 30. Our objective is to determine the optimal configuration that would yield the highest hit ratio and throughput.

We applied the same methodology as before to assess the efficacy of bulk eviction on both DRAM and PMem. This allowed us to identify the most suitable configurations for optimizing our HyperCache.

The findings, shown in Figure 5.7 and Figure 5.8, revealed that evicting approximately five segments at a time yielded the highest performance. It was observed that both larger or smaller bulk eviction sizes had lower performance.

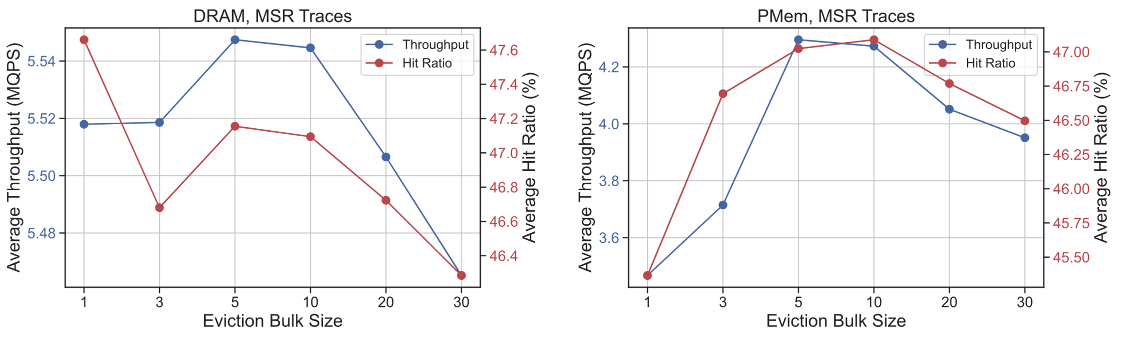

Figure 5.7: MSR Bulk Size Comparison

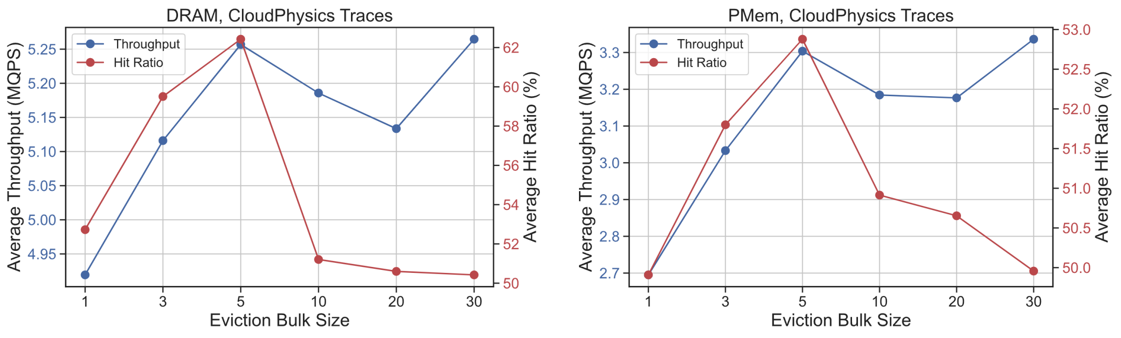

Figure 5.8: CloudPhysics Bulk Size Comparison

This pattern aligns with our expectations, and it offers two noteworthy insights:

- Bulk size too small:
  When the bulk eviction size is insufficient, the cache becomes filled more frequently, subsequently triggering a greater number of evictions. This frequent eviction process can negatively impact cache performance and diminish overall efficiency.
- Bulk size too large:
  Conversely, when the bulk eviction size is excessively large, each eviction operation takes a longer time to complete. Consequently, this prolonged eviction process can lead to performance degradation and negatively impact the system's throughput.

5.4 Hash Table Partition

While much effort has been devoted to reducing the DRAM usage of data objects, it's important to note that the hash table itself also consumes a significant portion of DRAM. Traditional hash table designs, while effective for many applications, can face challenges in this high-demand context. This concern becomes particularly significant as the number of elements required for data center caching grows.

For instance, ChatGPT-3.5 and ChatGPT-4 operate with the number of parameters from billions to trillions [57, 58]. Assuming each parameter uses 8 bytes only in the hash table by Segcache's design [2], this would consume terabytes of DRAM only for the hash table.

	chatGPT-3.5	**chatGPT-4**
Parameters	175 billion [57]	1 trillion [58]
Hashtbable Size (approx.)	1.4 terabytes	8 terabytes

Table 5.1: Hashbucket Usage in First Slots

However, not all data within the hash table is accessed frequently. We observed that certain parts of the hash table are accessed less frequently, suggesting an opportunity for optimization.

As a result, we propose a novel hybrid hash table design specifically tailored for our HyperCache, which more aggressively saves DRAM usage by partitioning the hash table and migrating a portion of it to PMem.

In this section, we will discuss the design of our hash table, the rationale behind the design, and the experimental data that supports the design.

5.4.1 Design Overview

Our design is built upon the Segcache [2] hash table design, which was described in Chapter 2, Section 2.5.2.

The hybrid hash table divides each hash bucket into two distinct partitions: the "hot" partition and the "cold" partition.

The hot partition is stored in DRAM and comprises three slots for each hash bucket: (1) A shared info slot containing the metadata about its hash bucket, which is necessary for every lookup operation; (2) A first object slot that holds the first object info[1] in the hash bucket; (3) A tag slot that stores all the reduced hashtags of the remaining objects.

[1]Object info in the hash table is metadata of the object, such as its segmentID, offset, access counter, timestamp etc. Not the actual key-value pair.

The cold partition of the hash bucket, on the other hand, contains the remaining object info slots and is stored in PMem. This partition is accessed less frequently. By storing it in PMem, we can leverage the high capacity and cost-effectiveness of this type of memory.

The below diagram 5.9 illustrates this hybrid hash table design.

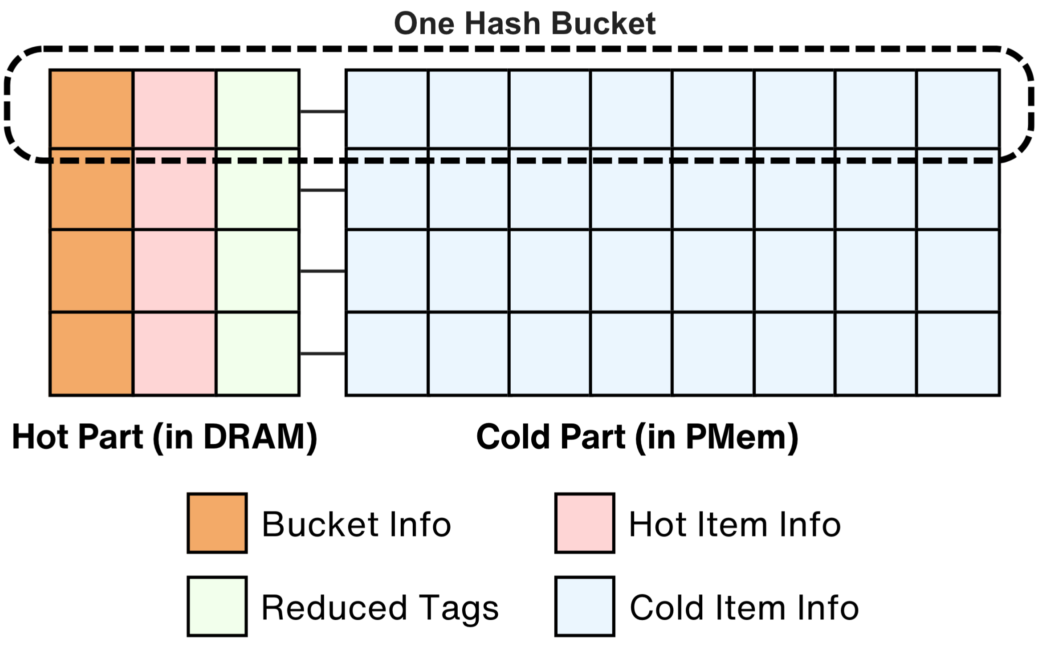

Figure 5.9: Hybrid Hash Table Design

A key innovation in our design is the use of reduced object tags. In the original Segcache design, each object slot contains a full tag. These tags help to reduce hash collision and therefore minimize unnecessary visits to the object-store in the cache when the tag doesn't match the lookup key.

In our design, we implement a strategy of compressing the reduced tags associated with the cold slots and concatenating them into a single slot within the hot. By examining the reduced tag in the compact slot, we can determine the cases when an object is not in the cold part. In essence, this compact reduced-tag slot functions as a bloom filter [59].

The algorithm block 5 in Appendix A provides a detailed illustration of the lookup process in our hash table design.

5.4.2 Design Rational

The design of our partitioned hash table is motivated and supported by empirical analysis. We collected slot usage patterns within hash buckets and the hash collision reduction effect of hash tags.

Hash Bucket Slot Usage Analysis

In this experiment, we processed two sets of traces to quantify the cumulative usage of each slot within the hash bucket. We determined the slots visited by each query and recorded the total number of visits for all slot indices across the entire set of traces.

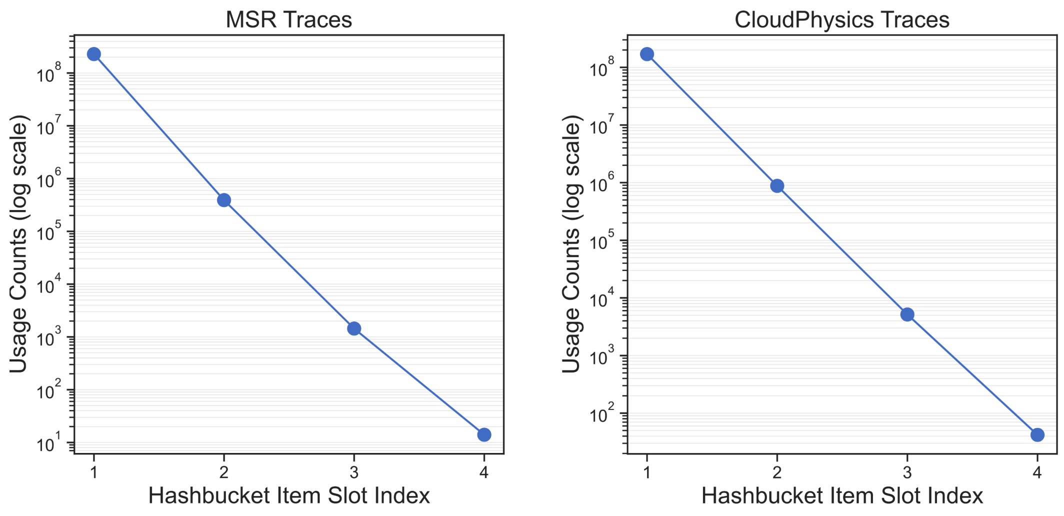

Figure 5.10: Hash Bucket Slots Usage

Traces	MSR	CloudPhysics
First Slot Visit	99.83%	99.48 %

Table 5.2: Hash Bucket Usage in First Slots

The experimental results, as shown in Figure 5.10 and Table 5.2, demonstrated a consistent trend: The first object slots in hash buckets are accessed significantly more than the remaining slots. Additionally, our examination of these traces indicated slots beyond index 5 are rarely used.

Hash Collision Analysis

In addition to slot usage, our design rationale also considered the impact of hash collisions on hash table performance. To investigate this, we conducted an experiment where we varied the length of hash tags and analyzed the resulting hash collisions. We compare the number of hash collisions with the total number of hash table queries to check the effect.

We selected five different lengths for the hashtags: 2, 4, 8, and 12. The results in 5.11

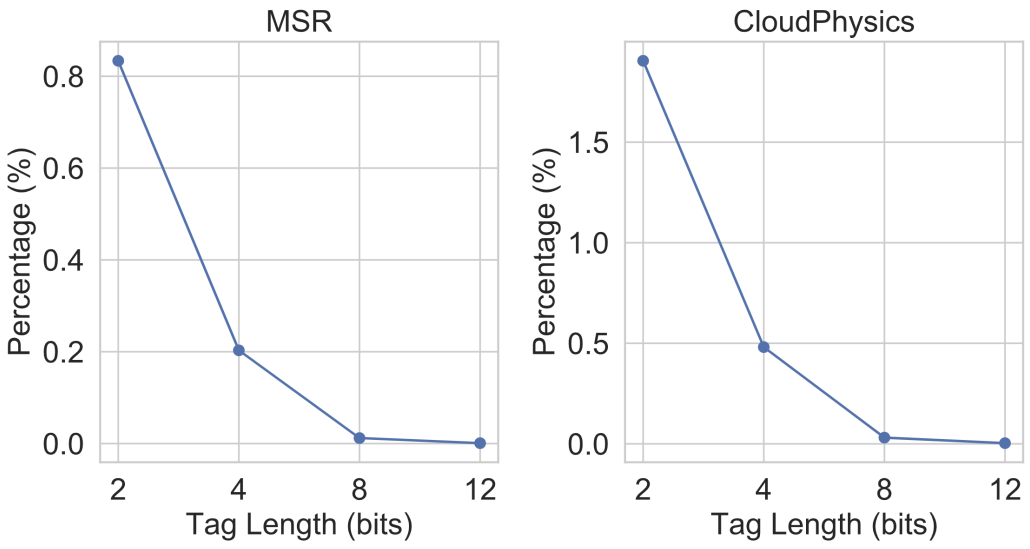

Figure 5.11: Hash Collision Analysis

showed that all lengths of hash tags helped to reduce hash collisions significantly, with longer tags having a better effect.

To compact the reduced tag into one hash bucket slot (8 bytes), we decided to use 8-bit per object as its reduced hash tag, and put it in DRAM to mitigate the footprint in PMem slots visits.

In summary, the rationale behind our design is rooted in two key observations: (1) A high frequency of access to the first object; (2) A much lower frequency of the rest of the object slots; (3) A reduced hash tag can still help to reduce hash collision significantly. Our design aims to address the issue of extensive DRAM consumption caused by the hash table, thereby optimizing the cost-efficiency of HyperCache.

Chapter 6

Evaluation Results

We first briefly discuss our final design after various refinements informed by careful analysis of real-world request traces. We then discuss the result of our evaluation. Our key aim is to demonstrate the improvement in DRAM efficiency, improvement in hit ratio, and the benfit of partitioning the hash table in hybrid memory.

6.1 Final Design: HyperCache

After careful analysis and a series of experiments, we present our final design, HyperCache. This system is a refined hybrid extension over Segcache, utilizing the capabilities of both DRAM and PMem for optimized caching memory efficiency. It also introduces a uniquely tailored eviction policy based on Linear Regression and a well-tuned bulk size for efficient collective eviction.

HyperCache applies a novel approach to cache management, combining the strengths of Segcache's Time-To-Live (TTL) grouping with the high-capacity attributes of DRAM and PMem. It is designed to address the challenges faced in contemporary data centers, such as the need for larger, faster, and more cost-effective cache systems.

While we initially considered incorporating a partitioned hash table into the design, this feature was ultimately made optional. The potential reason being, the hash table is a frequently visited spot in the cache, and relocating part of it to PMem resulted in a significant decrease in throughput. The potential overhead incurred from partitioning operations also proved detrimental to overall system performance.

In the following sections, we present a thorough performance evaluation of the HyperCache system, showcasing the effectiveness of our design decisions. The evaluations are conducted in the identical setup as described in Chapter 4. To visualize the distributions of the results from multiple traces, we utilize box plots with percentiles at 25%, 50%, and 75%. The whiskers of the box plots represent the 10% and 90% percentiles.

6.2 Enhancement in DRAM Efficiency

We first evaluate the efficiency of DRAM utilization within HyperCache. To do this, we vary the DRAM usage of HyperCache to be 10%, 30%, 50%, and 70% of Segcache's DRAM usage, and we examine how the performance of HyperCache changed in response. The comparison baselines are Segcache that uses 100% DRAM, and Segache that uses 100% PMem.

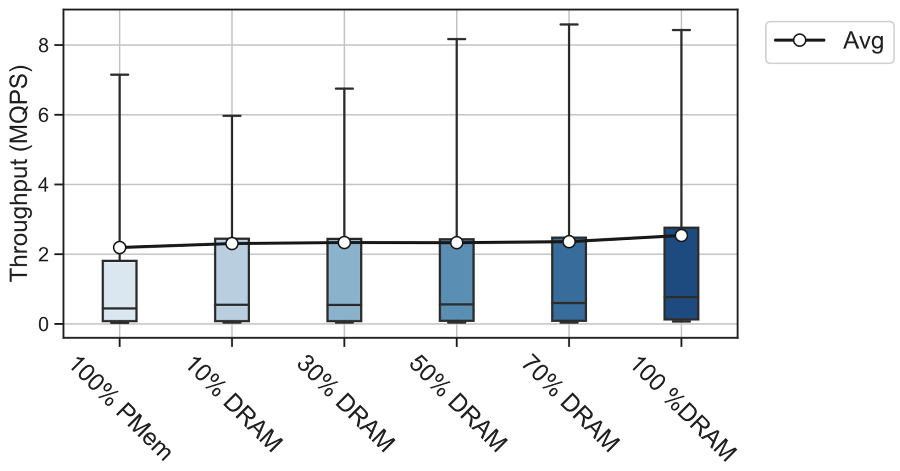

Figure 6.1: Throughput Distribution for Different DRAM Usages, MSR Traces

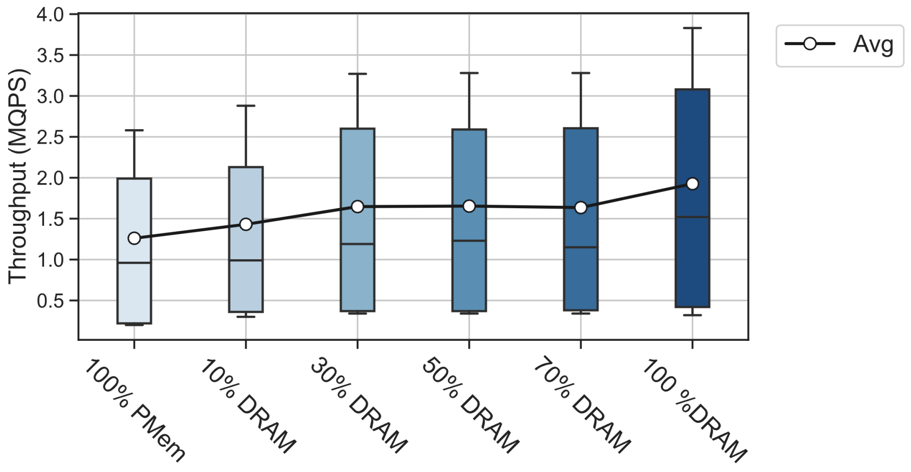

Figure 6.2: Throughput Distribution for Different DRAM Usages, CloudPhysics Traces

Remarkably, as shown in Figure 6.1 and Figure 6.2, HyperCache's performance closely mirrored that of an all-DRAM cache. The throughput differential of HyperCache and Segcache remains within 20%, despite HyperCache having **DRAM usage reduced by up to 70%**. This result demonstrates the effectiveness of our hybrid extension over Segcache, clearly showing that HyperCache is able to optimize DRAM usage without substantially impacting throughput. The implications of this finding are particularly noteworthy for data centers, where efficient utilization of resources is crucial.

6.3 Hit Ratio Improvement

Next, we assessed HyperCache's performance in terms of hit ratio, which is a key performance indicator for caching systems. With the same amount of DRAM as Segcache, HyperCache managed to **increase the hit ratio by 4.5%**.

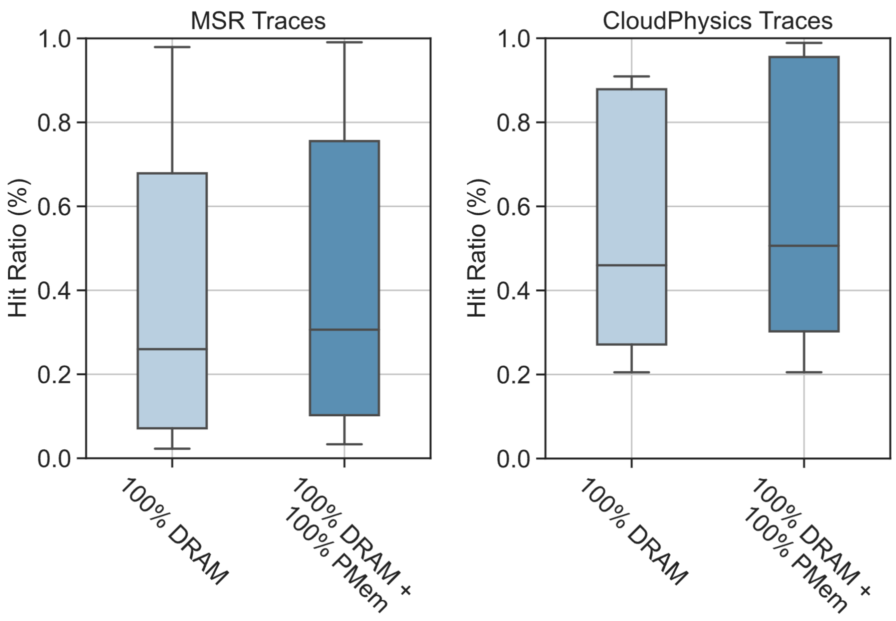

Figure 6.3: Hit Ratio Distribution of Segcache and HyperCache Using Same DRAM

Traces	PMem Hashtable	Hybrid Hashtable
MSR	44.34%	48.87%
CloudPhysics	57.34%	61.87%

Table 6.1: Average Hit Ratio of Segcache and HyperCache Using Same DRAM

This improvement in hit ratio, as in Figure 6.3 and Table 6.1, indicates that HyperCache is more efficient in managing cached data, leading to fewer cache misses and thus better overall performance. The enhanced hit ratio directly translates into faster responses for data requests, making HyperCache a highly efficient cache solution.

6.4 Benefit of Partitioned Hash Table

To evaluate the performance of the hash table, we use a baseline where all the hash table is stored in PMem. To solely assess the behavior of PMem with the hash table, we conduct the evaluation wherein all object-store segments are placed in DRAM. This allows us to observe the performance of PMem specifically in relation to the hash table without the influence of other factors.

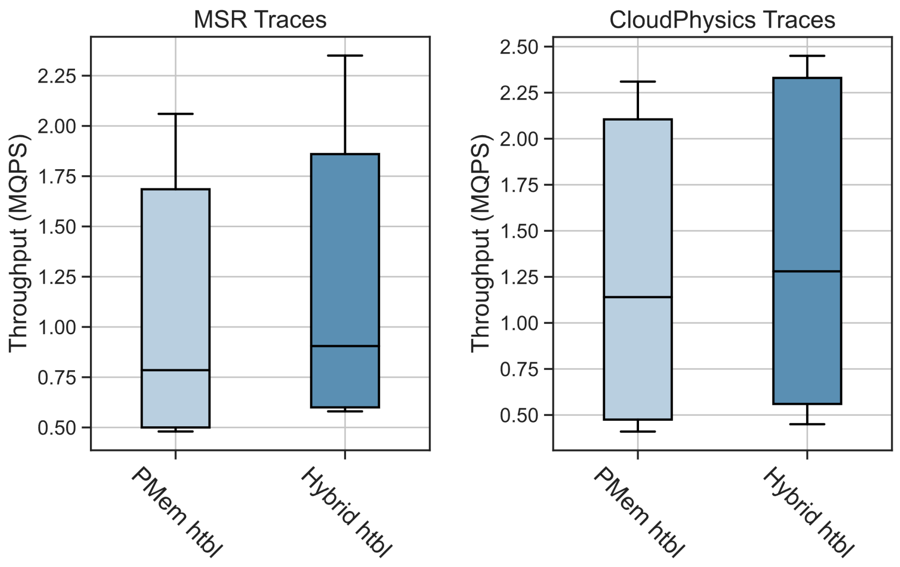

Figure 6.4: Throughput Distribution of PMem Hash Table and Hybrid Hash Table

Traces	PMem Hash Table	Hybrid Hash Table
MSR	1.95 MQPS	2.15 MQPS
CloudPhysics	1.43 MQPS	1.58 MQPS

Table 6.2: Average Throughput of PMem Hash Table and Hybrid Hash Table

Figure 6.4 and Table 6.2 show that compared to the baseline, the cache throughput has a significant improvement, with average throughput improved by around 6.78% and 10.49%. This offers the opportunity to allocate a portion of the hash table in PMem and reduce memory consumption.

However, we have observed that the throughput of the PMem hash table and the partitioned hash table is lower compared to the conventional DRAM hash table. This observation is based on the examination of the rightmost entry in Figure 6.1 and Figure 6.2. As discussed in the previous section 6.1, this performance degradation could be attributed to the frequent access of the entire hash table. Consequently, any computational overhead and latency induced by PMem are magnified due to the sensitivity of the hash table modification. In our future work, we can address this issue and explore more practical approaches for partitioning the hash table.

Chapter 7

Conclusion and Future Work

In this chapter, we discuss key findings from our experience of building and evaluating HyperCache. Additionally, we outline potential directions for future research, including exploring different eviction policies and further optimizing the use of hybrid memory in caching. This chapter highlights the significance of our research in the field of high-capacity caching systems and sets the stage for further advancements in this area.

7.1 Conclusion

We have now presented the design, implementation, and evaluation of HyperCache, a novel hybrid extension on top of Segcache that optimizes caching efficiency and throughput and is targeted at scaling for modern data centers.

We provide a set of design principles informed by careful analysis of cache eviction for real-world traces. These principles contribute to the optimization of caching-based service tiers. These principles include the proficient usage of DRAM and the reduction of footprint in PMem or any secondary layer memory device. These principles can serve as a guide to improve the performance and efficiency of caches for modern online services.

Our thesis shows the advantages and challenges of deploying hybrid caches, particularly their potential for cost-effective memory utilization and capacity benefits. We also introduce PMem to a broader range of applications, thereby expanding its utility and potential impact. Furthermore, our work stands at the forefront of applying machine learning in low-level computer systems, with many opportunities for further development.

7.2 Limitations

While HyperCache has shown promising results, there are some limitations and incompleteness in our work. These include:

- Lack of generality:
  The cache design presented in this study is specifically tailored to Segcache and ran on a specific hardware platform. We also primarily focus on adapting PMem, and the configurations used were not general to other types of devices and cache systems. These specificities limit the generality of our findings.

- Lack of comparison with other caches:
  The current study does not compare HyperCache with other in-production caches, such as Redis and Memcached. Future research should include such comparisons to provide a more comprehensive evaluation of HyperCache's performance relative to other widely used caching systems.

- Performance issue of hybrid hash table:
  The performance of the hybrid hash table was found to be lower than expected. This suggests a need for further investigation and optimization to enhance its efficiency and effectiveness.

- Load balancing of concurrent tasks:
  The current study does not focus on load balancing different concurrent tasks, such as expiration, eviction, and machine learning tasks. Future work could explore strategies for effectively balancing these tasks to optimize overall system performance.

By addressing these limitations in future research, we can continue to refine and enhance the design and performance of HyperCache, thereby contributing to advancements in the field of high-capacity caching systems.

7.3 Future Work

Today, memory and storage technology are developing at a rapid pace. The importance of our work in shaping future cache implementations becomes evident. In this context, we have identified six key areas for expanding our research. These areas not only build upon our existing work, but also aim to tackle emerging challenges and opportunities in the field. They cover a diverse range of topics, such as extending HyperCache's applicability to different hardware architectures like NVMe SSDs and disaggregated memory, leveraging machine learning in broader cache contexts, and exploring optimization techniques such as compression for small objects and adaptive strategies.

7.3.1 Cache over NVMe SSDs

Traditionally, SSDs have been placed behind a slow interface for ease of integration into disk-based servers. The SSD technology is rapidly evolving, and today's SSDs that use the NVMe protocol are much faster than their old (SATA-based) counterparts. Our work focuses on exploiting PMem for mitigating DRAM pressure in order to optimize caching for small objects, but growing a cache over block SSDs storage involves new challenges.

First, SSDs are block devices, which means that the content stored inside is accessed at a page granularity. Therefore we must group related objects together not just based on TTL, but also based on access patterns to reduce read/write amplifications. If a small object access brings an entire page of unrelated data into the OS's (Operating System's) I/O cache[1], then performance can degrade compared to a DRAM-only baseline. Furthermore, we must find a way to not pollute the I/O cache due to unrelated data. Growing caches over SSDs also brings forward a methodological challenge. How to best size the I/O cache in relevant to the user heap storing cached data is not obvious. In essence, we must manage the I/O cache efficiently for data center caches. Few prior works looked into the issues of DRAM partitioning between user-level cache and OS cache, and our future work can deal with the problem in the context of data center caches for small objects.

7.3.2 Cache over Disaggregated Memory

Disaggregated (remote) memory offers several advantages compared to local memory and storage. It enables remote memory to be shared among multiple applications, allowing for efficient utilization based on the memory demands of each application. With recent advancements in networking technology, we believe there are three competing and sometimes complementary approaches to expanding main memory capacity: (1) Local flash storage; (2) New memory technologies, such as phase change memory or PMem; (3) Remote memory. These technologies offer different trade-offs across multiple dimensions, including price, performance, and capacity.

Growing a cache over remote memory poses similar challenges as expanding a cache over SSDs or storage, but it also introduces new considerations. More specifically, one must decide what background operations to run on the remote server with limited computation capacity. Tasks such as merging and defragmenting caches can require significant computational resources. These operations can either be offloaded to a remote server, or discarded to reduce the computational burden, at the cost of memory inefficiency. We leave it to future work to evaluate specific mechanisms and policies for growing a cache over a combination of local and remote memory.

[1]Prior literature also refer to I/O cache also as page cache or buffer cache

7.3.3 OS Caching Impacts and Opportunities

As discussed in the previous subsection 7.3.1 "Cache over NVMe SSDs", proper management of the OS cache is essential when expanding the cache over local storage. In addition, a hybrid system with byte-addressable DRAM and PMem can also employ the OS cache by using a special mount-time feature of modern file systems. In this work, we use PMem in direct-access mode, bypassing the OS page cache. In our future works, we can explore the potential benefits of enabling OS caching in PMem, local storage, and remote memory deployments of HyperCache. One specific challenge will be determining the appropriate size of the OS cache in relation to the overall DRAM capacity. Additionally, we can investigate whether the default OS cache eviction algorithm requires tuning to efficiently operate with HyperCache.

7.3.4 Compression for Small Objects

Compression is another key technique data-centric applications use to conserve memory usage and improve memory efficiency. Compressing small objects is particularly challenging, as the entropy of the cached data is high. Prior works have only infrequently used compression in caching and key-value stores due to the challenge of finding efficient compression algorithms for small objects. We believe that a deep knowledge of underlying data stored in a cache can open up insights and opportunities for the use of compression in HyperCache and similar caches for small objects.

We are particularly interested in compression as a knob depending on the type of environment cache is deployed. For example, suppose a large number of integers are cached, then in this case, we can utilize integer compression algorithms (that have advanced a lot in recent times due to their ubiquitous presence in online services). We can further optimize memory efficiency by incorporating support for a wide range of compression algorithms and selecting the most suitable one based on the operator's understanding of the environment. More importantly, we can save memory-related expenditure, which is a crucial factor in the overall cost of modern data-center deployments.

7.3.5 Adaptive Eviction Strategies

In this study, we have focused on dynamic eviction algorithms while statically selecting the ML algorithms and hyperparameters for a specific cache. This has provided us with valuable insights into the advantages of dynamic adaptation strategies.

Data center cache operates around the clock for very long periods of time. An even more robust strategy for long-running workloads is to adapt the caching management algorithm on a phased basis. It is well-known that caches go through phase behavior, which is primarily influenced by user behaviors over time. For example, the cache system for Twitter can exhibit hot-spots on specific topics during special events or on particular days, giving a spike in demand for tweets related to those topics.

We believe we can construct a more flexible and dynamic system that can change its configurations and eviction algorithms automatically. A potential solution is to employ a "tournament" framework where multiple eviction policies compete with each other. Furthermore, we can also utilize reinforcement learning agents to choose the eviction policies, building upon prior research [10, 51] as mentioned in the Related Work Chapter 3, Section 3.4.2. To achieve efficient auto-tuning of hyperparameters and configurations, we can adopt similar ML approaches used in "Competitive Caching with Machine Learning Advice" [60]. However, effectively balancing these additional algorithms with the workload of the caching system poses its own set of challenges. In future works, we can conduct a comprehensive analysis of the underlying motivations as an extension of HyperCache.

7.3.6 ML Eviction for General Caches

Lastly, within the scope of this work, we have primarily focused on TTL-based caches, which are common in today's data centers. However, it is worth noting that other types of caches, such as Redis and Memcached, use different insertion and grouping strategies. We can apply the insights from this work to improve the performance and efficiency of generalized non-TTL-based caches. We can also study software stacks closely related to caches, such as key-value stores. Although these applications do not employ eviction, they still require substantial memory capacity. Given the continuous expansion of user data within online service infrastructures, the benefits offered by the approaches outlined in this thesis will become increasingly valuable in large-scale cache deployments and similar services.

Appendix A

Appendix: Algorithm Pseudo-codes

Algorithm 1: Segcache Read Operation

```
procedure READ(key)
    tag ← hashtag(key)
    bucket ← hashTable.getBucket(tag)            ▷ Go to corresponding hash bucket
    for each slot in bucket do
        if slot.tag == tag then                  ▷ Compare the tag with slots in hashtable
            segment ← segments.get(slot.segID)
            item ← segment.getItem(slot.offset)
            if item.key == key then              ▷ Compare the key
                if item is not expired then
                    return item.value            ▷ Cache hit
                end if
            end if
        end if
    end for
    return null                                  ▷ Cache miss
end procedure
```

Algorithm 2: Segcache Write Operation

```
1: procedure WRITE(key, value, ttl)
2:     bucket ← ttlBuckets.getBucket(ttl)
3:     segment ← bucket.getTailSegment()
4:     if segment is full then
5:         if Cache is full then                        ▷ If there are no free segments
6:             Evict()                                  ▷ Perform eviction
7:         end if
8:         segment ← allocateNewSegment()
9:         bucket.addSegment(segment)
10:        Write(key, value, TTL)                       ▷ Re-attempt after eviction
11:    end if
12:    segment.appendItem(key, value)
13:    hashTable.insert(key, segment)                   ▷ Update hashtable
14: end procedure
```

Algorithm 3: Segcache Expire Operation

```
1: procedure EVICT
2:     for each bucket in ttlBuckets do
3:         segment ← bucket.getTailSegment()
4:         while segment is not null do
5:             if segment.creationTime + segment.ttl $\leq$ currentTime then
6:                 bucket.removeSegment(segment)        ▷ Segment is expired
7:                 for each s in (bucket.head ... segment) do
8:                     freeSegment(s)                   ▷ All the parent segments are also expired
9:                 end for
10:                break
11:            end if
12:            segment ← segment.prevSegment
13:        end while
14:    end for
15: end procedure
```

Algorithm 4: HyperCache Write Operation

```
procedure WRITE(key, value, ttl)
    $bucket$ ← ttlBuckets.getBucket(ttl)
    $DRAMsegment$ ← bucket.getTailDRAMSegment()
    if $DRAMsegment$ is full then
        if DRAM is full then
            if PMem is full then
                EvictFromPMem()                          ▷ PMem also full
            end if
            EvictDRAMToPmem()
        end if
        $DRAMsegment$ ← allocateNewSegmentInDRAM()
        bucket.addSegment($DRAMsegment$)
        Write(key, value, ttl)                           ▷ Re-attempt after eviction
    end if
    $DRAMsegment$.appendItem(key, value)
    hashTable.insert(key, $segment$)
end procedure
```

Algorithm 5: Hybrid Hashtable Read Operation

```
$H_b$ : hot slot of hashbucket $b$
$C_b$ : cold slot array of hashbucket $b$
$T_b$ : reduced tags associated with cold slots of hashbucket $b$

procedure READ($key$)
    $t$ ← hashtag($key$)
    $b$ ← hashTable.getBucket($t$)                   ▷ Go to corresponding hash bucket
    if $H_b$.tag == $t$ then
        return Segments.get($H_b$)                   ▷ Cache hit
    else                                             ▷ Item is not in the hot slot
        for $i$ in len($C_b$) do
            if $T_b[i]$ == reduced($t$) then
                return Segments.get($C_b[i]$)        ▷ Cache hit
            end if
        end for
    end if
    return $null$                                    ▷ Cache miss
end procedure
```

Bibliography

[1] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. Characterizing the performance of intel optane persistent memory: A close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, page 488–505, New York, NY, USA, 2022. Association for Computing Machinery. [Cited on pages 2 and 12.]

[2] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 503–518. USENIX Association, April 2021. [Cited on pages 2, 3, 14, 15, 16, 23, 32, and 39.]

[3] Zhenyu Song, Daniel S. Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, Santa Clara, CA, February 2020. USENIX Association. [Cited on pages 2, 25, and 33.]

[4] Matteo Fiorani, Slavisa Aleksic, Paolo Monti, Jiajia Chen, Maurizio Casoni, and Lena Wosinska. Energy efficiency of an integrated intra-data-center and core network with edge caching. *J. Opt. Commun. Netw.*, 6(4):421–432, Apr 2014. [Cited on page 7.]

[5] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 167–181, New York, NY, USA, 2013. Association for Computing Machinery. [Cited on page 7.]

[6] Vijay Kumar Adhikari, Sourabh Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting youtube: An active measurement study. In *2012 Proceedings IEEE INFOCOM*, pages 2521–2525, 2012. [Cited on page 7.]

[7] Amazon Web Services. Database caching strategies using redis, 2023. [Cited on page 7.]

[8] Nivedita Mishra, Sharnil Pandya, Chirag Patel, Nagaraj Cholli, Kirit Modi, Pooja Shah, Madhuri Chopade, Sudha Patel, and Ketan Kotecha. Memcached: An experimental study of ddos attacks for the wellbeing of iot applications. *Sensors*, 21(23), 2021. [Cited on page 7.]

[9] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, San Francisco, CA, March 2003. USENIX Association. [Cited on pages 9 and 25.]

[10] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354. USENIX Association, February 2021. [Cited on pages 9, 25, and 53.]

[11] Kathlene Morales and Byeong Kil Lee. Fixed segmented lru cache replacement scheme with selective caching. In *2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC)*, pages 199–200, 2012. [Cited on page 9.]

[12] Sabrina Müller, Onur Atan, Mihaela van der Schaar, and Anja Klein. Context-aware proactive content caching with service differentiation in wireless networks. *IEEE Transactions on Wireless Communications*, 16(2):1024–1036, 2017. [Cited on page 9.]

[13] Jeroen Famaey, Frédéric Iterbeke, Tim Wauters, and Filip De Turck. Towards a predictive cache replacement strategy for multimedia content. *Journal of Network and Computer Applications*, 36(1):219–227, 2013. [Cited on page 9.]

[14] Suoheng Li, Jie Xu, Mihaela van der Schaar, and Weiping Li. Trend-aware video caching through online learning. *IEEE Transactions on Multimedia*, 18(12):2503–2516, 2016. [Cited on page 9.]

[15] Sara A. Elsayed, Sherin Abdelhamid, and Hossam S. Hassanein. Predictive proactive caching in vanets for social networking. *IEEE Transactions on Vehicular Technology*, 71(5):5298–5313, 2022. [Cited on page 9.]

[16] Bruce Jacob. *Memory systems: Architecture and design*. Elsevier, 2015. [Cited on page 11.]

[17] Onur Mutlu. The rowhammer problem and other issues we may face as memory becomes denser. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1116–1121, 2017. [Cited on page 11.]

[18] Dramexchange - world leading dram and nand flash market research firm, with more than a decade of most authoritative database. [Cited on page 11.]

[19] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *SIGMETRICS Perform. Eval. Rev.*, 37(1):181–192, jun 2009. [Cited on page 12.]

[20] Sparsh Mittal, Jeffrey S. Vetter, and Dong Li. A survey of architectural approaches for managing embedded dram and non-volatile on-chip caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, 2015. [Cited on page 12.]

[21] Intel Corporation. Intel® optane™ persistent memory, 2023. [Cited on pages 12 and 24.]

[22] Twitter. Pelikan cache, 2023. [Cited on pages 14 and 21.]

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. SIGMETRICS '12, page 53–64, New York, NY, USA, 2012. Association for Computing Machinery. [Cited on page 14.]

[24] Juncheng Yang, Yao Yue, and K. V. Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208. USENIX Association, November 2020. [Cited on page 14.]

[25] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast In-Memory Key-Value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association. [Cited on page 15.]

[26] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *2018 ACM SIGMOD International Conference on Management of Data (SIGMOD '18), Houston, TX, USA*. ACM, June 2018. [Cited on page 15.]

[27] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery. [Cited on page 17.]

[28] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001. [Cited on page 19.]

[29] Bjoern H Menze, B Michael Kelm, Ralf Masuch, Uwe Himmelreich, Peter Bachert, Wolfgang Petrich, and Fred A Hamprecht. A comparison of random forest and its gini importance with standard chemometric methods for the feature selection and classification of spectral data. *BMC Bioinformatics*, 10(213), 2009. [Cited on page 19.]

[30] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The cachelib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, OSDI'20, USA, 2020. USENIX Association. [Cited on page 21.]

[31] Amazon Web Services. Amazon elasticache, 2023. [Cited on page 22.]

[32] Salvatore Sanfilippo. Redis, 2023. [Cited on page 22.]

[33] Brad Fitzpatrick. Memcached, 2023. [Cited on page 22.]

[34] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications. 17(4), oct 2021. [Cited on page 23.]

[35] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing dram footprint with nvm in facebook. EuroSys '18, New York, NY, USA, 2018. Association for Computing Machinery. [Cited on page 23.]

[36] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, SOSP '21, page 243–262, New York, NY, USA, 2021. Association for Computing Machinery. [Cited on page 23.]

[37] Abhishek Rajimwale, Vijayan Prabhakaran, and John D. Davis. Block management in solid-state devices. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 21, USA, 2009. USENIX Association. [Cited on page 23.]

[38] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-performance transactions for persistent memories. *SIGPLAN Not.*, 51(4):399–411, mar 2016. [Cited on page 24.]

[39] Youyou Lu, Jiwu Shu, and Long Sun. Blurred persistence: Efficient transactions in persistent memory. *ACM Trans. Storage*, 12(1), jan 2016. [Cited on page 24.]

[40] Iulian Moraru, David G. Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan Binkert. Consistent, durable, and safe memory management for byte-addressable non volatile main memory. TRIOS '13, New York, NY, USA, 2013. Association for Computing Machinery. [Cited on page 24.]

[41] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. Thynvm: Enabling software-transparent crash consistency in persistent

memory systems. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 672–685, 2015. [Cited on page 24.]

[42] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 401–410, New York, NY, USA, 2012. Association for Computing Machinery. [Cited on page 24.]

[43] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: A tiered file system for Non-Volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, Boston, MA, February 2019. USENIX Association. [Cited on page 24.]

[44] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 460–477, New York, NY, USA, 2017. Association for Computing Machinery. [Cited on page 24.]

[45] Vaishnav Janardhan and Adit Bhardwaj. Predictive Caching@Scale. Santa Clara, CA, May 2019. USENIX Association. [Cited on page 25.]

[46] Daniel S. Berger. Towards lightweight and robust machine learning for cdn caching. In *Proceedings of the 17th ACM Workshop on Hot Topics in Networks*, HotNets '18, page 134–140, New York, NY, USA, 2018. Association for Computing Machinery. [Cited on page 25.]

[47] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. Feedforward neural networks for caching: N enough or too much? 46(3):139–142, jan 2019. [Cited on page 25.]

[48] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. [Cited on page 25.]

[49] Jerome H. Friedman. Greedy function approximation: A gradient boosting machine. *The Annals of Statistics*, 29(5):1189 – 1232, 2001. [Cited on page 25.]

[50] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* The MIT Press, 2018. [Cited on page 25.]

[51] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, Boston, MA, July 2018. USENIX Association. [Cited on pages 25 and 53.]

[52] Song Jiang and Xiaodong Zhang. Lirs: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. SIGMETRICS '02, page

31–42, New York, NY, USA, 2002. Association for Computing Machinery. [Cited on page 25.]

[53] Dongsheng Yang, Daniel S. Berger, Kai Li, and Wyatt Lloyd. A learned cache eviction framework with minimal overhead. 2023. [Cited on page 26.]

[54] Juncheng Yang, Ziming Mao, Yao Yue, and K. V. Rashmi. GL-Cache: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 115–134, Santa Clara, CA, February 2023. USENIX Association. [Cited on pages 26 and 33.]

[55] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. Write off-loading: Practical power management for enterprise storage. 4(3), nov 2008. [Cited on page 27.]

[56] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, Santa Clara, CA, February 2015. USENIX Association. [Cited on page 27.]

[57] OpenAI. Generative pre-trained transformer 3 models, 2020. [Cited on page 39.]

[58] OpenAI. Generative pre-trained transformer 4, 2023. [Cited on page 39.]

[59] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. 13(7):422–426, jul 1970. [Cited on page 40.]

[60] Thodoris Lykouris and Sergei Vassilvitskii. Competitive caching with machine learned advice. *J. ACM*, 68(4), jul 2021. [Cited on page 53.]