Psyche: A Resource-Efficient TTL-Aware In-Memory Cache Policy

by

David Chu

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto

© Copyright 2025 by David Chu

PSYCHE: A Resource-Efficient TTL-Aware In-Memory Cache Policy
David Chu
Master of Applied Science
The Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto
2025

Abstract

Modern web-scale systems rely on in-memory caches to provide users with the nearly instantaneous response time that they expect, but that persistent storage, such as hard-disk drives, cannot deliver. In addition to traditional capacity-based *evictions* (such as with LRU or LFU) due to memory constraints, modern in-memory caches must handle time-to-live (TTL) expiries to ensure data freshness. However, supporting both these policies adds metadata overhead and compute requirements, while the cost of not supporting expiries is memory overuse from caching expired data.

Currently, there are three methods for handling expiries: 1) discard expired objects lazily when they are accessed (Lazy-TTL), 2) discard expired objects proactively by maintaining an expiry queue, in addition to an eviction queue (Proactive-TTL), or 3) discard expired objects periodically by performing a scanning or sampling operation (Periodic-TTL). Lazy-TTL wastes memory by caching expired objects, Proactive-TTL adds cache metadata overhead, and Periodic-TTL adds computation overhead and increases the number of expiry-related probing accesses.

We introduce PSYCHE, a novel predictive caching policy that handles both eviction and expiry policies while being metadata- and compute-efficient. Its key innovation is predicting whether the eviction or the expiry policy will first remove an object, thus allowing objects to be cached in only one queue. To do so, Psyche analyzes cache evictions to create a profile of the temporal characteristics of the eviction policy, which enables maintaining a *predicted eviction time*. When caching an object, PSYCHE compares the known TTL of the object with the predicted eviction time to determine whether to place the object in the eviction or expiry queue. Psyche lowers memory usage compared to Lazy-TTL, lowers metadata overhead compared to Proactive-TTL, and lowers computation overheads compared to Periodic-TTL.

We evaluate PSYCHE on production Twitter cache traces, demonstrating that it matches the miss ratio performance of Proactive-TTL, while reducing eviction/expiry queue metadata by 50%, or 16 bytes per object, which is significant for small object caches. Compared to Periodic-TTL, we reduce the number of objects accessed by the expiry policy by up to 371× compared to Memcached while still discarding expired objects quickly.

To my maternal grandfather, Lawrence Yee, who inspired me to pursue engineering.

Acknowledgements

Thank you to those who encouraged and enabled me to pursue higher education. This includes my family, friends, teachers, and colleagues. I am particularly grateful for the unwavering support of my two supervisors, Professor Ashvin Goel and Professor Michael Stumm, and for the thoughtful insights of my committee members, Professor Angela Demke Brown, Professor Ding Yuan, and my committee's chair, Professor Zeb Tate.

Contents

1 Introduction **1**

1.1 Contributions 3

1.2 Organization 3

2 Background **4**

2.1 Eviction Policies 4

2.1.1 Single-Queue Eviction Policies 4

2.1.2 Multi-Queue Eviction Policies 6

2.2 Time-to-Live (TTL) Expiry Policies 7

2.2.1 TTL Semantics 8

2.3 Miss Ratio Curves 8

2.3.1 Generating Miss Ratio Curves 8

2.4 Existing In-Memory Caches 11

2.4.1 Memcached 11

2.4.2 Redis 11

2.4.3 CacheLib 12

2.4.4 SegCache 12

2.4.5 PaperCache 13

3 The Psyche Design **14**

3.1 Psyche Data Structures 15

3.2 Predictions with PSYCHE 17

3.2.1 When to Predict 17

3.2.2 Estimating Eviction Time 18

3.2.3 Initial Estimation of Eviction Time 19

3.2.4 Handling Mispredictions 19

3.2.5 Handling Changing Workloads 21

3.3 PSYCHE Eviction Policies 22

3.3.1 LRU/PSYCHE 22

3.3.2 LFU-f/PSYCHE 22

3.4 Discussion 23

4 Evaluation **24**
4.1 Experimental Dataset 24
4.2 Experimental Method 25
4.2.1 Simulating Periodic-TTL Policies 25
4.2.2 Miniature Simulations 26
4.3 Evaluation Results 27
4.3.1 Miss Ratio 27
4.3.2 Memory Usage 27
4.3.3 Probing Accesses 29
4.4 Sensitivity Analysis 29
4.4.1 Alternate Eviction Time Statistics 29
4.4.2 Dynamic Workloads 29
4.4.3 LFU Approximation 30

5 Conclusions **40**
5.1 Future Work 40

Bibliography **42**

List of Tables

1.1 Comparison of TTL-based caching schemes. Lower is better. 2

2.1 Redis's eviction policies. 12

3.1 Terminology. 16

4.1 Recommended Twitter traces, as per Yang et al. 25
4.2 An example trace where LRU/TTL violates the inclusion principle. 26
4.3 An example of two caches where LRU/TTL violates the inclusion principle. 26

List of Figures

3.1 TTL versus position in LRU caches (1 GiB and 4 GiB) for the Twitter Cluster #19 workload. The coloured lines show how an object may travel within a cache. 15
3.2 An LRU/Proactive-TTL cache. 16
3.3 Proactive-TTL and Psyche object data structures. LFU requires an additional frequency counter in both data structures. 17
3.4 Mispredictions in Psyche. 20

4.1 Miss ratio curves for LRU/Proactive-TTL, LRU/Psyche and TTL-only policies. . . 31
4.2 Miss ratio curves for LFU/Proactive-TTL, LFU-1/Proactive-TTL, and LFU-1/Psyche policies. 32
4.3 Memory usage for the LRU/Proactive-TTL and the LRU/Lazy-TTL caching policies for the Twitter Cluster #19 workload. 33
4.4 LRU/Psyche's memory savings over LRU/Proactive-TTL. 34
4.5 LFU-1/Psyche's memory savings over accurate LFU/Proactive-TTL. 35
4.6 Memory usage of Periodic-TTL policy compared to Psyche. 36
4.7 Object accesses by the expiry policy for Memcached, CacheLib and Psyche. 37
4.8 The miss ratio curves and memory savings for various predicted eviction times for a 4 GiB cache for the Cluster #19 workload. 38
4.9 The changes in the eviction time for Cluster #19 using a 1 GiB cache and a 4 GiB cache. 38
4.10 LRU/Psyche's memory savings compared to LRU/Proactive-TTL for varying levels of decay for Cluster #19. 39
4.11 Increasing the number of logical LRU queues increases the LFU-approximation accuracy for Cluster #52. 39

Chapter 1

Introduction

In-memory caches are widely used to accelerate the retrieval of data in large scale systems. Data normally resides in a cheap but slow storage medium, such as a hard-disk drive whose latency and throughput cannot meet the demands of modern web-scale applications. For this reason, data is often cached in a faster caching medium, such as DRAM, which provides much higher throughput with lower latency.

Caches are more expensive per byte compared to storage and so they typically contain only a subset of the total working set. Deciding what to keep and what not to keep in the cache remains an open research question [6, 30, 31, 19]. The most commonly deployed eviction policy is the least recently used (LRU) eviction policy [7, 16, 2], whereby the caches evict objects in reverse order of access recency to make room for incoming objects that are not in the cache. In LRU, an object is placed at the head of the queue when it is newly inserted or when it is accessed. This means a dormant object will be pushed toward the tail of the queue. Eviction occurs from the tail when a newly accessed object needs to be inserted and the cache is full. Another common eviction policy is the least frequently used (LFU) eviction policy [16, 19, 2], which evicts objects that have been accessed the fewest number of times first. When there is a tie, LFU evicts the least recently used amongst the tied objects.

Modern in-memory caches often use time-based expiries, called time-to-live (TTL) attributes. Each object is assigned an *expiry time* when it is inserted in the cache.[1] An object expires and therefore becomes unservable to the client when the current time exceeds the expiry time (i.e., TTL is ≤ 0). There are several uses of TTLs in caches. First, they can provide a time bound on data inconsistency [28]. Second, they can be used to periodically refresh data or run infrequent but expensive computations [28]. Third, they can be used for more efficient cache sizing based on the unexpired working set size [21]. Finally, they can be used for automatic deletion, such as for privacy reasons [5]. For example, the European Union's General Data Protection Regulation (GDPR) stipulates that its citizens have a right to be forgotten without undue delay, generally considered to be about a month [27].

Modern caches must efficiently handle TTL expiries in a resource-efficient manner. The simplest option for handling expired objects is to discard them lazily when they are accessed, which we call *Lazy-TTL*. This approach may retain expired objects in the cache for a long period of time until

[1] There are other variants of TTL expiries, but we use this one throughout this work.

TTL Expiry Policy	Relative Miss Ratio	Nr. of Resident Expired Objects	Additional Metadata Overhead for TTL Policy	Nr. of Probing Accesses
TTL-only	High	Low	Low	Low
Lazy-TTL	Low	High	Low	Low
Proactive-TTL	Low	Low	High	Low
Periodic-TTL	Low	Medium to Low	Low	High
Psyche	Low	Low	Low	Low

Table 1.1: Comparison of TTL-based caching schemes. Lower is better.

they are either accessed sometime in the future or eventually evicted. Since expired objects in the cache cannot be served, they simply waste cache space and thus unnecessarily increase cache memory requirements. In fact, prior work has shown that Lazy-TTL can use so much memory for expired objects that it becomes impractical [21].

A second approach for handling TTL expiries, used by caches such as SegCache [29] and Paper-Cache [19], is to maintain a dedicated data structure such as a TTL expiry queue for proactively discarding objects as they expire or soon after they expire. This approach, which we call *Proactive-TTL*, avoids or minimizes keeping expired objects in the cache, thus reducing cache memory requirements, but it requires additional memory for the TTL-related cache metadata.

With memory being the most critical resource for caches, popular industrial-grade caches, such as Memcached, Redis, and CacheLib, opt for a third approach that avoids maintaining any dedicated TTL structures. Instead, they periodically search the cache by either scanning the entire cache or sampling a subset of the cache to remove expired objects [8, 17, 12]. This approach, which we call *Periodic-TTL*, avoids any TTL-related cache metadata overhead, but it caches expired objects until the next search operation and the search adds computation and memory access overheads.

We propose PSYCHE,[2] a caching method for handling both cache evictions and expiries efficiently. Our key idea is to use two queues, an *eviction queue* and an *expiry queue*, and place objects in one or the other based on a prediction of which queue will trigger removal of the object. When our predictions are accurate, PSYCHE will avoid caching expired objects, reduce TTL metadata overhead compared to Proactive-TTL, and avoid or minimize the need for periodic search operations for expired objects compared to Periodic-TTL.

Our predictor estimates the likelihood of an object expiring before it is evicted. When an object is evicted from the eviction queue, we record the real time that the object has taken to traverse the eviction queue (e.g., from the head to the tail of the queue), which we call the *eviction time*. We maintain a histogram of these eviction times and use the median eviction time to estimate the current *predicted eviction time* for the cache. When an object is cached, PSYCHE compares the known TTL time of the object with the predicted eviction time. This comparison is possible because both the eviction time and object TTL times are based on real time. If the TTL time is smaller, then the object is placed in the expiry queue, since it is more likely to expire before it is evicted. Otherwise, it is placed in the eviction queue.

Table 1.1 shows a qualitative comparison of various TTL-based caching schemes along four metrics: cache miss ratio (ratio of cache misses relative to cache accesses), memory overhead due to expired objects, TTL-metadata memory overhead, and memory access overhead. The TTL-only caching scheme uses the expiry queue for eviction as well, which our evaluation shows increases the

[2]Psyche was given the task of sorting an enormous amount of grain in classical mythology.

miss ratio. Existing policies are efficient for only three of the four metrics, while PSYCHE uses its prediction mechanism to achieve efficiency for all four metrics.

We implemented PSYCHE for the LRU and LFU cache eviction policies. Our evaluation demonstrates that PSYCHE'S predictive method of placing objects in the eviction or expiry queue can resolve the trade-off between memory overheads and computation overheads in current TTL-aware caches. We evaluate PSYCHE on production Twitter cache traces and show that it matches the miss ratio performance of Proactive-TTL. Compared to Proactive-TTL, we reduce eviction/expiry queue metadata by 50%, or 16 bytes per object, which is significant for small object caches, and yields up to 700 MiB savings on a cache size of 8 GiB. Compared to industrial-grade caches using Periodic-TTL, such as Memcached and CacheLib, we reduce the number of accesses needed to expire objects by a geometric mean of 8.9× and 454× respectively, over the production traces released by Twitter.

1.1 Contributions

This thesis makes the following contributions:

- We analyze the behavior of various TTL-based eviction policies and show that they make different trade-offs in terms of caching efficiency, memory overhead or computation overhead.
- We propose the PSYCHE caching policy that handles both cache evictions and expired objects efficiently by correlating the temporal behavior of the eviction policy with TTL expiries.
- We implement and evaluate PSYCHE using the LRU and LFU eviction policies and show up to 8% total memory savings on small-object caches compared to Proactive-TTL policies and up to 134 000 times fewer expiry-related accesses compared to existing, deployed caches.

1.2 Organization

This dissertation is organized as follows: Chapter 2 discusses background and related work. Chapter 3 describes the design of PSYCHE for the LRU and the LFU eviction policies. Chapter 4 evaluates Psyche by comparing it with existing TTL-based eviction policies. Finally, Chapter 5, provides our conclusions and discusses avenues for further research.

Chapter 2

Background

This chapter explores prior work related to in-memory caches. In-memory caches are a type of cache where the main dataset resides on a slow medium, such as a hard disk drive, and hot (heavily-accessed) data is stored in a faster medium, such as DRAM, to provide better performance. In-memory caches are typically implemented as key-value stores, where an object consists of a key and a value, and may have metadata associated with it. We discuss cache eviction policies in Section 2.1, expiry policies in Section 2.2, cache simulation techniques in Section 2.3 and then describe well-known or highly-deployed in-memory caches in Section 2.4.

2.1 Eviction Policies

Caches have a finite size and so when full, they need to evict an object to make room for new objects. The exact choice of which object to evict is called the cache *eviction policy*. The eviction policy typically uses some metadata to store historical information about objects' access patterns and creates a relative ordering that determines the order in which the objects are evicted.

In this section, we first describe single-queue eviction policies (LRU, LFU, FIFO, Second-Chance and Sieve [31]) and then more complex multi-queue policies (S3-FIFO [30], and 2Q [6]). In practice, implementing eviction policies accurately has memory or performance overheads (e.g., exact LRU is hard to parallelize) and so some real world implementations modify the eviction algorithms for increased performance or lower memory overhead [7, 16].

A related caching concept is *admission policies*. Instead of deciding which object to remove from the cache when it is full, an admission policy decides which objects may enter the cache. An effective admission policy reduces the number of objects that are evicted. Admission policies can be coupled with eviction policies. Admission policies include algorithms such as TinyLFU [3] and 2Q [6]. TinyLFU, for instance, keeps probabilistic counters of all object reference counts and only admits new objects if they have been accessed previously.

2.1.1 Single-Queue Eviction Policies

LRU, LFU, Second-Chance, and Sieve are popular eviction policies that use a single queue to rank the objects from hottest to coldest.

Least Recently Used (LRU)

The most popular eviction policy is the least recently used (LRU) eviction policy. Popular in-memory cache implementations, such as CacheLib [2], Memcached [7], and Redis [16], support this policy. LRU orders objects based on their access recency, with the most recently accessed object located at the head of the queue and the least recently accessed object at the tail of the queue. New objects are inserted at the head of the queue. When an object is reaccessed, it is also moved to the head of the queue. When the cache needs to make space, it will remove the eponymous least recently used object from the tail of the queue.

While LRU is a popular eviction policy due to its conceptual simplicity, it has notable drawbacks. First, in a multithreaded implementation, LRU needs to lock the head on each access, which limits scalability and performance. Second, LRU is not scan resistant, so if an application scans through a dataset and populates a cache with those objects, it will push out other objects in the cache, even if these new objects will never be accessed again [30].

Least Frequently Used (LFU)

Another popular eviction policy, supported by CacheLib [2] and Redis [16], is the least frequently used (LFU) eviction policy. This policy works by keeping track of the number of times each object has been accessed, which it calls the *frequency*. When the cache needs to make room for new objects, it will choose the least frequently used object, breaking ties with the least recently used object with the lowest access frequency. By evicting objects primarily based on access frequency, LFU is able to resist scanning patterns better than LRU. However, with changing workloads, LFU has no mechanism to downgrade previously frequently accessed objects, so it may evict a useful object with a low access count rather than a cold object with a high access count [3].

There are multiple variants of LFU. Ideal LFU tracks the frequency of every object ever seen. Even if an object is evicted from the cache, its frequency counter is still tracked. Ideal LFU consumes memory proportional to the total number of unique objects ever seen, rather than the number of unique objects in the cache, which can be prohibitively expensive for long-running caches. Practical LFU only tracks the frequency of objects in the cache. When an object is inserted into the cache, it is initialized with a frequency of 1 regardless of whether it had been in the cache previously.

There exist methods that approximate the full history of an object's access frequency by using memory-efficient probabilistic counters. TinyLFU uses this method to implement an admission policy. It requires incoming objects to have been accessed a certain number of times before they are admitted into the cache [3]. However, one could couple this probabilistic technique for out-of-cache objects with exact counters for in-cache objects to closely approximate ideal LFU.

First In, First Out (FIFO)

First-in, first-out (FIFO) inserts objects at the head of the queue and evicts objects from the tail, in the same order as their insertion, regardless of the reuse pattern of objects in the cache. This algorithm performs well on flash because it reduces the number of metadata writes, since accessing an object does not affect its position in the eviction queue [30]. Also, FIFO makes it simple to determine the order in which objects will be evicted.

The main drawback of FIFO is that popular objects that have been accessed many times in the

past, and will continue to be accessed in the future, may eventually be evicted when they become the oldest object in the cache, which can lead to popular objects being evicted. Methods to address this drawback include Second-Chance, Sieve [31], and S3-FIFO [30]. Another drawback of FIFO caches is that they can be hard to size because they experience a phenomenon known as Bélády's Anomaly, in which increasing the size of the cache does not guarantee the miss ratio will remain the same or decrease.

Second-Chance

Second-Chance is a modification of FIFO that takes access patterns into account. When an object in the cache is accessed (i.e., reaccessed), it is marked as visited. During eviction, an object is evicted from the tail, similar to FIFO, but only if it is not marked visited. Otherwise, the object is unmarked, and moved to the head of the queue, as if it were a newly inserted object. This technique of saving a visited object is sometimes referred to as *lazy promotion* [30]. Second-Chance approximates LRU without incurring the cost of moving objects to the head of the queue when they are accessed. The Second-Chance algorithm can be implemented more efficiently with a circular queue, which is called Clock.

Sieve

Sieve [31] is similar to Second-Chance except that eviction happens at a location in the queue called the *hand* that is initialized to the tail of the queue. During eviction, if the hand encounters an unvisited object, the object is evicted, similar to Second-Chance. Otherwise, the object is unmarked, and the hand moves one object closer to the head. The hand moves back to the tail when it encounters a visited object at the head. The advantage of Sieve is that frequently accessed objects tend to collect at the tail and remain in the cache. Its disadvantage is that it tends to be outperformed by LFU and S3-FIFO [19].

2.1.2 Multi-Queue Eviction Policies

2Q [6] and S3-FIFO [30] are popular eviction policies that use multiple queues to separate hot objects from cold objects. These policies introduce key-only queues, which are queues that only store the key of an object. This allows these policies to track the access pattern of a greater number of objects without needing to store all the objects.

2Q

2Q [6] consists of three queues, the *A1-in* FIFO queue, the *A1-out* key-only FIFO queue, and the *main* LRU queue. A1-in uses 25% of memory, A1-out uses enough memory to store the keys whose objects would fill half the cache, and main uses the rest. New objects are inserted into the A1-in queue. When an object reaches the end of the A1-in queue, the object's value is evicted, but its key is placed in the A1-out queue. An object that is accessed while its key is in the A1-out queue is admitted into the main LRU queue. The A1-in and A1-out queues collectively serve as an admission policy for the main LRU queue. Memcached uses a variant of 2Q [7] and CacheLib supports a variant of 2Q [11]. 2Q's primary innovation is related to its admission queue: it filters scanned objects but forces objects to be evicted from memory before they are re-admitted to the main queue.

S3-FIFO

S3-FIFO uses three queues to filter hot objects from cold objects. The three queues are a *small* probationary FIFO queue, a *main* Second-Chance queue, and a *ghost* FIFO key-only queue for recently evicted objects. S3-FIFO determines the behaviour of objects by tracking their visit counts in a two-bit saturating counter that is incremented on each access. Newly inserted objects start as unvisited and are placed at the head of the small queue, which comprises roughly 10% of the cache. As objects exit the small queue, S3-FIFO places visited objects into the main queue, which comprises about 90% of the cache, and decrements their visit count by one. The main queue is like a Second-Chance queue, except it can record up to 3 visits due to the two-bit counter. It decrements the count by one each time the object moves to the head. Unvisited objects are evicted when they exit the main queue. Finally, unvisited objects' keys move to the ghost queue when they exit the small queue. The ghost queue can hold the same number of keys as the main queue, but contains no values. An object whose key is in the ghost queue will be inserted directly into the main queue and its key removed from the ghost queue. Upon exiting the ghost queue, a key is simply forgotten and will be treated as new if it is encountered again. Compared to LRU, S3-FIFO's lock-free design allows it to scale better and its small probationary queue makes it more resistant to scans.

2.2 Time-to-Live (TTL) Expiry Policies

Time-to-live (TTL) cache *expiry policies* are a family of policies used to put a time limit on how long an object may remain valid in the cache. There are several reasons for using TTL policies. First, proactively removing expired objects drastically reduces the working set size [21]. Second, cached data may become stale if an object is updated at the storage system and the cached object is not invalidated [13]. Thus, applications may choose to maintain data freshness in the cache by assigning time-to-live attributes to objects so that the objects eventually expire and are discarded. Finally, TTL policies enable satisfying privacy policies that require stale objects to be discarded after a certain period. For these reasons, TTLs are mandated in some cases, such as at X (formerly Twitter) [28]. They are typically set heuristically with the domain expertise of the engineer.

The *Proactive-TTL* policy maintains a dedicated expiry queue in addition to an eviction queue. An object's metadata will reside in both queues so that it can be removed from the cache by way of an eviction or an expiry. However, only one of these policies will cause the removal of the object and so storing metadata in both queues wastes memory resources. Thus, *Psyche* aims to maintain metadata for each object in either the eviction or expiry queue.

The expiry queue adds metadata costs, e.g., per-object pointers for maintaining the queue. The *Periodic-TTL* policy avoids using an expiry queue. Instead, it periodically searches for expired objects in the cache. However, this approach adds compute requirements and increases the number of probing accesses. The number of probing accesses accounts for the search cost (e.g., number of key accesses) needed to expire objects. Moreover, there are diminishing returns of the number of expired objects found by increasingly aggressive search strategies.

The *Lazy-TTL* policy removes expired objects when they are accessed. Lazily discarding expired objects avoids the costs of an active policy, but it wastes more memory storing expired objects.

TTLs can also be used as a heuristic for an eviction policy. Since TTLs are set using the domain expertise of the engineer, and objects with TTLs near their expiry time are infrequently accessed [29],

an eviction policy can evict the soonest expiring object regardless of whether this object has expired, which we call the *TTL-only* policy (Redis calls it the volatile TTL policy [16]).

2.2.1 TTL Semantics

There are different use cases for TTLs and so TTLs can have different meanings in different systems. Here, we discuss the semantics of refresh and hardness.

An object's TTL may refresh upon each access, i.e., the TTL is reassigned based on a new expiry time. The advantage of refreshing a TTL on access is that hot objects do not time out as frequently. The disadvantage is that with Proactive-TTL eviction, the ordering of objects constantly changes as objects are accessed, which would make certain modern caches, such as Segcache [29], that depend on the TTL ordering not changing frequently, impractical.

A TTL is hard when an object has no utility once it has expired; in other words, it must not be served to the user. Conversely, an object with a soft TTL may be served to users after it has expired, but the cache may take some action to deal with the soft expiry. Furthermore, objects may have both a soft and hard TTL, where the soft TTL typically expires before the hard TTL.

We assume non-refreshing, hard TTLs because these are the standard semantics used in the in-memory caching community and in the Twitter traces [28].

2.3 Miss Ratio Curves

A common metric for measuring the efficacy of a cache is a *miss ratio*, which is the ratio of requests that cannot be serviced by the cache (misses) to the total number of cache requests. The miss ratio may change over time but generally an eviction policy with a lower overall miss ratio for the entire workload is considered better. Note that some literature refers to a related but inverse concept, the hit ratio [18], in which a higher hit ratio is better.

One of the fundamental design considerations in a cache is the miss ratio versus cache size trade-off. A larger cache size typically results in a lower miss ratio because it stores more objects. A *miss ratio curve(MRC)* shows the miss ratio of a cache as a function of the cache size. Miss ratio curves are an effective tool for comparing eviction policies, and for sizing caches, i.e., choosing a minimum cache size while maintaining an acceptable miss ratio [21].

We will use miss ratio curves to compare PSYCHE with other cache eviction schemes. In this section, we discuss efficient methods for generating miss ratio curves, which we will use for simulating caches in our evaluation in Chapter 4.

2.3.1 Generating Miss Ratio Curves

The most intuitive method of generating a miss ratio curve is by simulating a cache with different sizes using a trace of cache accesses (i.e., access trace) and recording the miss ratio for each size. This method is expensive because it requires a pass through the access trace for each cache size that is simulated. In following text, we describe three types of algorithms, stack algorithms, approximate stack algorithms and non-stack algorithms, for generating miss ratio curves.

Stack Algorithms

Some eviction policies adhere to the *inclusion principle*, which states that for a given access trace, an object contained in a smaller cache will always be contained in every larger cache. The inclusion principle implies that the objects in the cache can be ordered in a stack and smaller caches represent a smaller view (subset) of this stack. Thus, such eviction policies are also called *stack algorithms*. The miss rate curves of stack algorithms can be generated efficiently for all cache sizes in a single pass of the trace because when an object is evicted from a cache of a certain size, it must also be evicted from all smaller caches [9].

The Mattson algorithm [9] generates MRC for the LRU eviction policy for all cache sizes in a single pass. The main idea is to maintain an ordered list of objects by access recency and record in a histogram the *stack distance* of an object when it is accessed. The stack distance is defined as the distance from the head of the LRU stack (where new objects are inserted) to the object that is being accessed; or if the object has never been accessed before, then it is defined as infinite. This distance is the number of other unique items accessed since the previous access to this item. The stack distance helps determine whether the access would result in a cache hit or a miss for a given cache size. If the cache size is larger than the stack distance, the access is a hit (otherwise, a miss), which exploits the inclusion principle. Thus, the cumulative distribution function of the stack distance histogram is the hit ratio curve `Hit(cache_size)` of the eviction policy. The miss ratio curve is the complement `Miss=1-Hit` of this function. For stack algorithms, the miss ratio curve is a strictly decreasing function with cache size.

Algorithm 1 shows the pseudocode for Mattson's algorithm. It has a time complexity of $O(N \times M)$, where N is the number of accesses and M is the number of unique items in the access trace. The space complexity is $O(M)$. In practice, the algorithm can be implemented with a singly linked list with a single pointer per object.

Algorithm 1 Mattson's Single-Pass MRC Generation Algorithm

```
 1: procedure MATTSON(trace, hash_table, lru_list, histogram)
 2:     for current_time, key in enumerate(trace) do
 3:         if key in hash_table then
 4:             prev_time ← hash_table[key]
 5:             stack_dist ← lru_list.stack_dist(prev_time)
 6:             lru_list.pop(prev_time)
 7:         else
 8:             stack_dist ← ∞
 9:         end if
10:         hash_table[key] ← current_time
11:         lru_list.push(current_time)
12:         histogram[stack_dist] ← histogram[stack_dist] + 1
13:     end for
14:     return 1 − cdf(histogram)
15: end procedure
```

Olken improved on Mattson's algorithm by implementing the LRU stack as a balanced tree, with $O(N \times \log M)$ time complexity [14]. The space complexity remains $O(M)$. In practice, a balanced tree uses two pointers per object, which increases memory requirements.

Approximate Stack Algorithms

Generating an exact MRC for stack algorithms is expensive because the fastest known algorithm has $O(N \times \log M)$ time and $O(M)$ space complexity. In production workloads, N and M can be large, especially for long-running, large caches. There exist many approximate techniques for MRC generation that significantly reduce complexity over exact algorithms. We discuss a few below.

Spatially Hashed Approximate Reuse Distance Sampling (SHARDS) [24] is a sampling technique that reduces the number of keys that are processed in the access trace. SHARDS's key idea is to use consistent sampling to select keys from a small but consistent subset of all the keys. It does so by hashing every key and comparing the hash to a threshold; if the hash is less than the threshold, then it is sampled. SHARDS typically processes approximately one key per hundred or one key per thousand, with reasonable accuracy.

SHARDS has two variants, fixed-rate and fixed-size. Fixed-rate SHARDS chooses to sample an object by hashing the object and selecting it if it is beneath some fixed threshold, which means that on average, SHARDS samples keys at a fixed-rate. The time complexity is still $O(N \times \log M)$ to process the access trace and the space complexity is $O(M)$, where N and M are the number of sampled accesses and unique sample objects, respectively. While the Big-O complexity remains the same, the multiplication of N and M by small constants due to sampling means that in practice, fixed-rate SHARDS performs very well. Fixed-size SHARDS is able to maintain fixed space by maintaining a moving threshold. This variant removes sampled items with the largest hash value and lowers the threshold over time to sample a fixed number of objects. Therefore, its time complexity is still $O(N \times \log M)$, but its space complexity is $O(1)$.

AET is a temporal approximation method for MRC generation [4]. It works by constructing a histogram of the *reuse time*, which is defined as the total number of accesses between accesses to key k. This is in contrast to the stack (or reuse) distance, which measures the total number of *unique* objects accessed between accesses to key k. AET defines average eviction time as the average number of accesses that occur before the key is evicted from a cache of a certain size. It is calculated from the reuse time histogram by solving an implicit equation based on certain observations of an LRU cache. Then, the approximate MRC is generated based on the probability that a key has a reuse time greater than the average eviction time. The time complexity of AET is $O(N)$ and the space complexity is $O(1)$.

AET's average eviction time concept may seem similar to Psyche's eviction time but AET's is based on number of accesses whereas Psyche uses real time for the predicted eviction time, which makes it possible to compare with TTLs and implement expiry efficiently. Moreover, AET models average eviction times based on the speed of movement of objects in an LRU cache, which doesn't account for objects expiring due to TTLs, or other eviction policies such as LFU. In contrast, Psyche directly uses eviction times for its prediction, which is more generalizable.

Non-Stack Algorithms

Non-stack-based algorithms do not guarantee the inclusion principle. That is, an object in a smaller cache may not exist in a larger cache. In fact, sometimes, they may exhibit Bélády's Anomaly, whereby the miss ratio is not strictly decreasing for larger cache sizes [1]. Non-stack-based algorithms include FIFO, Practical LFU [20], Second-Chance, Sieve, S3-FIFO, and 2Q. Here, we discuss two

methods for generating approximate MRC for non-stack algorithms.

Miniature Simulations (Minisim) [23] simulates caches of different sizes while applying the fixed-rate SHARDS technique, thereby reducing the number of objects represented in the simulation. Stack algorithms with proactive non-refreshing TTLs (such as LRU/Proactive-TTL) are also non-stack algorithms. As a result, we use Minisim extensively throughout this work for our evaluation.

Kosmo [20] is a more memory efficient technique than Minisim. It simulates multiple caches at a time and tracks the eviction of objects in eviction records for each cache size. Kosmo is able to prune many eviction records by approximating all cache policies, so they follow the inclusion principle. While this introduces a small amount of error, Kosmo uses an average of 3.6 times less memory that Minisim.

2.4 Existing In-Memory Caches

In this section, we describe notable in-memory caches, namely Memcached, Redis, CacheLib, Seg-Cache, and PaperCache. We briefly summarize how they handle object eviction and expiry.

2.4.1 Memcached

Memcached is a widely-adopted, open-source in-memory cache. By default, it uses a segmented approximate-LRU eviction policy, a modified version of 2Q [8, 7]. Memcached also supports the LRU eviction policy. Objects are grouped by approximate size in slab classes. When the cache runs out of memory, it will evict an object from the same slab class as the incoming object.

Memcached allows multithreaded execution by reducing the movement of objects in the segmented approximate-LRU queues. Unlike LRU that eagerly promotes objects when they are accessed, Memcached delays promotion until the object reaches the tail of the segmented LRU queue. This approximation is similar to Second-Chance except that Memcached decides whether to promote an object based on the object attaining at least two accesses during its stay in the queue.

2.4.2 Redis

Redis is a memory-optimized, open-source in-memory cache. It uses various memory-saving optimizations to support approximations of the eight eviction policies shown in Table 2.1. The allkeys eviction policies may evict any object; the volatile eviction policies only evict objects whose expire field is set [16].

Redis stochastically approximates the LRU and LFU eviction policies by picking k objects at random. The default value for k is 5. It compares the last access time or the access frequencies, respectively, of each of the k objects, and then evicts the LRU or LFU of the k objects. This optimization reduces memory overhead since it does not need to store an ordered list of objects. However, this approach can produce a higher miss ratio. Increasing the value of k improves the miss ratio, but it also increases sampling overhead, e.g., generating k random numbers, and thus decreases throughput [25].

Eviction Policy	Description
noeviction	Stop caching new objects when cache is full.
allkeys-lru	Remove objects starting with the least recently used (LRU).
allkeys-lfu	Remove objects starting with the least frequently used (LFU).
allkeys-random	Remove objects in a random order.
volatile-lru	Remove objects with the expire field set to true in LRU order.
volatile-lfu	Remove objects with the expire field set to true in LFU order.
volatile-random	Remove objects with the expire field set to true in a random order.
volatile-ttl	Remove objects with the expire field set to true starting with the object with least time-to-live (TTL).

Table 2.1: Redis's eviction policies.

2.4.3 CacheLib

CacheLib is a library for creating caches on both DRAM and flash that was developed at Facebook. CacheLib is designed to be extensible with respect to eviction policies. It was created to simplify the myriad bespoke caches developed at Facebook by unifying these caches into a single library. Early versions of CacheLib did not improve on the existing custom caches (and sometimes even led to performance regressions). However, the lowered maintenance burden of a single caching system led to its adoption by Facebook [2]. CacheLib supports LRU (and variants), 2Q, and TinyLFU (a variant of LFU) natively, and it allows additional eviction policies to be created [2].

2.4.4 SegCache

SegCache [29] is an in-memory cache developed at Carnegie Mellon University and is used in production by X (formerly Twitter). It makes a number of design decisions based on the assumption that objects will primarily be removed from the cache due to TTL expiries. It allocates memory by grouping objects into segments based on similar TTLs, so when an object expires, it is likely that other objects within the segment are also expired. Thus, an entire segment can be freed soon after the first object in the segment expires. We believe the expiry queue in PSYCHE could benefit from these allocation optimizations.

SegCache manages its cache size by using a variant of LFU that takes both frequency of access and size into account. Since objects are grouped into segments based on TTLs, it needs to regroup objects into new segments when it wants to evict a portion of a segment. SegCache is notable in that it explicitly prioritizes TTL expiries over the capacity-based eviction policy, versus other caches that have added TTL expiries to an existing cache. This comes at a greater cost of eviction due to the overhead of merging partially evicted segments.

2.4.5 PaperCache

PaperCache [19] is an in-memory cache developed at the University of Toronto. It is specifically designed for handling changing workloads by switching the eviction policy depending on which eviction policy performs best at the current time. PaperCache maintains both a full eviction queue (for the eviction policy that it is currently implementing) and a full TTL expiry queue. Each object in the expiry queue is expired individually, rather in bulk, as in SegCache. This cache motivates the need to efficiently support TTLs with multiple eviction policies, not just a modified version of LFU, as in SegCache.

Chapter 3

The Psyche Design

This chapter describes the design of PSYCHE, a predictive TTL-aware cache that handles cache evictions and expiries efficiently. Similar to the Proactive-TTL approach, PSYCHE maintains an eviction queue and an expiry queue. Both Proactive-TTL and PSYCHE use an *eviction policy* to evict objects from its eviction queue to limit memory usage, and both use an *expiry policy* to discard expired objects from its expiry queue. The eviction policy is triggered when a new object needs to be inserted and the memory usage reaches the maximum cache size, while the expiry policy is triggered periodically, in our case, once per second.

PSYCHE aims to emulate the behavior of Proactive-TTL, while reducing metadata overheads. Proactive-TTL maintains object metadata in both the eviction and the expiry queues. These policies operate independently and so either may cause an object to be removed from the cache. When an object is removed, it is deleted from both the eviction and the expiry queues. The key idea behind PSYCHE is that if we can predict by which policy an object will be removed, then we only need to place its metadata in the corresponding queue. When our prediction is correct, PSYCHE will mimic the Proactive-TTL policy while reducing metadata overheads.

Our predictor estimates the likelihood of an object expiring before it is evicted. To do so, we need to find a common measurement for expiry and eviction. Real time is a natural candidate since the expiry policy uses TTLs (based on expiry times) and we know these times when an object is inserted in the cache. Thus, we need to estimate an object's expected *eviction time*, or the time when an object will be evicted from the cache (assuming it has no TTL). We can then compare the expiry time and the expected eviction time to determine whether the object's metadata should be placed in the eviction or expiry queue.

Figure 3.1 provides intuition for our approach. We ran the Twitter Cluster #19 workload using the LRU/Lazy-TTL policy on two caches of size 1 GiB and 4 GiB. For both graphs, we show 1000 sampled objects that are equally spaced within their respective LRU queue, for the same snapshot of time. Each point represents an object and shows its TTL versus its position in the LRU queue. The thick bands contain many newly inserted objects and objects that are accessed only once. As an object ages, it tends to move towards the eviction line at tail of the LRU queue (vertical black dotted line) and toward the expiry line (horizontal black dashed line), which results in diagonal movement toward the lower-right of the graph. PSYCHE predicts the slope of this diagonal by measuring how long evicted objects took to travel from the head to the tail of the LRU queue, which we call the

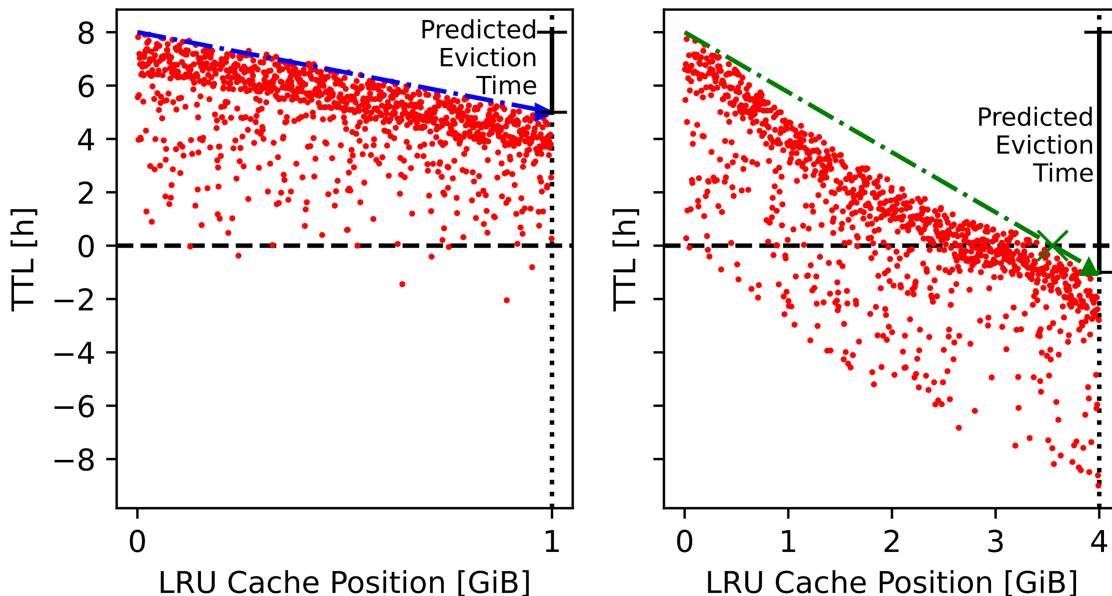

Figure 3.1: TTL versus position in LRU caches (1 GiB and 4 GiB) for the Twitter Cluster #19 workload. The coloured lines show how an object may travel within a cache.

predicted eviction time.

Psyche predicts whether an object will expire or be evicted by checking whether the diagonal line will intersect with the eviction line or the expiry line first. We show a hypothetical object's movement through time in both queues with the blue and green diagonal lines. Both objects are inserted into the cache with a TTL of 8 hours and move through the LRU queue similarly to other objects in the thick red band, which largely consists of objects inserted with a TTL of 6 to 8 hours and never reaccessed. The object following the blue line will be evicted because it intersects the eviction line first; the object following the green line will expire because it intersects the expiry line first (at the green X).

The rest of this chapter describes our approach in detail. Table 3.1 shows the terminology used in this chapter. Section 3.1 presents the data structures used by Psyche. Section 3.2 describes our prediction mechanism. Section 3.3 presents our implementation of LRU and LFU eviction polices in Psyche. We call the resultant policies LRU/Psyche and LFU/Psyche, respectively. Finally, Section 3.4 discusses current limitations and potential extensions to our design.

3.1 Psyche Data Structures

Figure 3.2 shows an example of an LRU cache implementing Proactive-TTL. The lookup table allows indexing cached objects by their keys. The eviction queue is implemented as a doubly-linked list to allow any object to be extracted from the queue, for example, when an object is reaccessed and promoted by the LRU eviction policy, or discarded by the expiry policy. The expiry queue is implemented as a (balanced) binary search tree so that objects stored in expiry time order can be discarded efficiently in $O(\log M)$ time.

In Psyche, we use a red-black tree for the expiry queue, but other sorted structures such as

Cache size	Maximum number of bytes that the cache can occupy.
Memory usage	Number of bytes used in the cache (size of all keys and values and metadata)
Eviction	When an object is removed from the cache due to cache size constraints.
Eviction policy	Policy used to evict objects to keep memory usage under cache size.
Expiry time	Real time when an object should no longer be served to the user.
TTL	Relative time from current time to expiry time.
Expiry	When an object outlives its TTL.
Discard	When an object is removed from the cache because it has expired.
Removal	When an object is evicted or discarded.
Expiry policy	Policy used to discard expired objects.
Accurate cache	Cache that maintains both full eviction and expiry queues so every object can be found in both queues.
Predictive cache	Cache that maintains partial eviction and expiry queues so an object is found exactly one queue.

Table 3.1: Terminology.

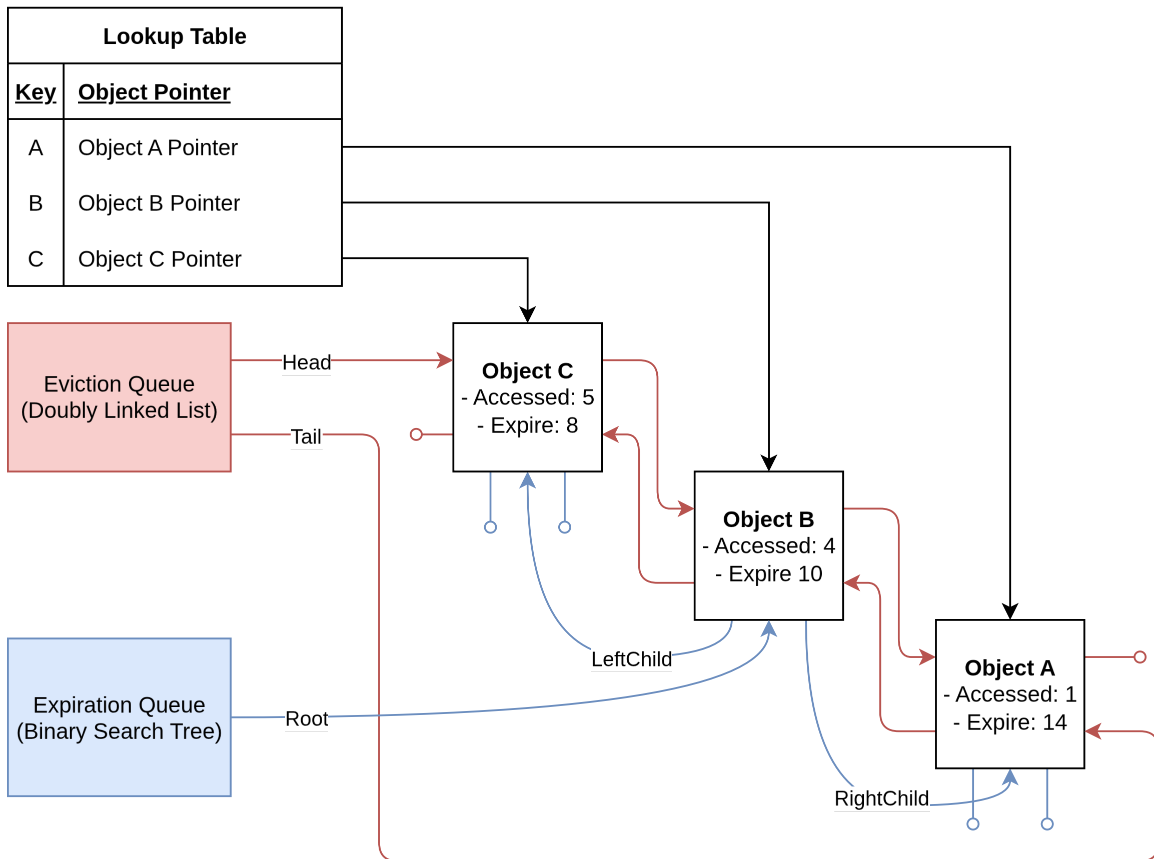

Figure 3.2: An LRU/Proactive-TTL cache.

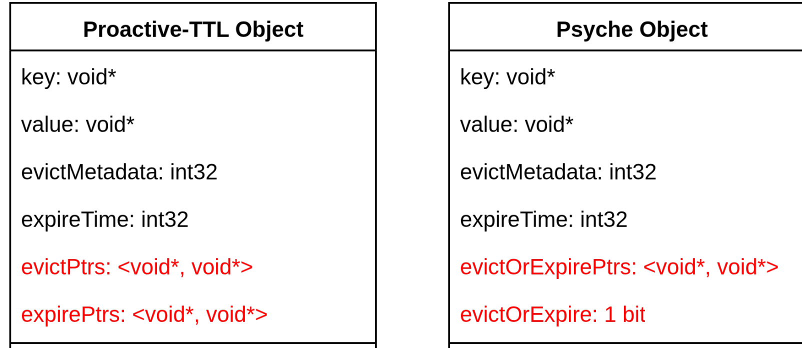

Figure 3.3: Proactive-TTL and Psyche object data structures. LFU requires an additional frequency counter in both data structures.

splay trees could be used as well. We do not use a skip list even though it can be more efficient because it has higher metadata requirements. We also do not use a heap because it does not support bulk expiry efficiently.

Psyche places an object in either the eviction queue or the expiry queue, thus eliminating one set of pointers, compared to Proactive-TTL. Figure 3.3 shows the object structures for Proactive-TTL and Psyche. The `evictMetadata` field is used to track eviction metadata such as the last-access time in LRU, access frequency and last-access time in LFU, and insertion time in FIFO. While this data may not be essential for an eviction policy, it is deemed sufficiently important that deployed caches such as Redis [15] and CacheLib [2] include this metadata. In this example, Psyche has 16 byte savings per object since it only stores one set of pointers, for either the eviction or the expiry queue. Psyche needs to store a bit indicating whether the object is in the eviction or expiry queue but this bit can be stored within a pointer, which uses 48 bits on a modern 64-bit system.

3.2 Predictions with Psyche

This section describes how Psyche predicts whether an object is more likely to expire or be evicted. Subsection 3.2.1 describes when Psyche makes predictions. Subsection 3.2.2 explains how we estimate eviction time to make predictions, and Subsection 3.2.3 presents how we bootstrap our estimate for a cold cache. Subsection 3.2.4 and Subsection 3.2.5 describe our method for handling mispredictions and changing workloads.

3.2.1 When to Predict

Psyche makes a prediction when it decides whether to place an object in the eviction or the expiry queue. Psyche makes a compulsory prediction on newly inserted objects. It makes a voluntary prediction when there is a strong signal that the prior prediction may no longer be a good prediction. For example, an object re-access is typically a strong signal that the object will remain in the cache longer in most eviction policies. As another example, a voluntary prediction occurs in Second-Chance during eviction when a visited object at the tail is reinserted at the head of the queue. Section 3.3 describes when we make predictions for the LRU and LFU policies. With changing workloads, the eviction times of objects may deviate from our predictions significantly, and so we can perform

additional voluntary predictions, as described in Subsection 3.2.5.

3.2.2 Estimating Eviction Time

This section describes how Psyche uses prior data to predict whether an object is more likely to expire or be evicted in the future. As mentioned earlier, we know an object's TTL and we compare this with an object's expected eviction time. However, it is not obvious how an object's eviction time should be estimated. Suppose we define eviction time as the time an object spends in the cache until eviction. The problem with this definition is that certain heavily-accessed (hot) objects will cause significant variations in the eviction time for non-FIFO policies. For example, some hot objects will stay in the cache for a long time. When they are eventually evicted, they will have a long eviction time. Moreover, when a new object is cached, we do not know whether it will be reaccessed, and so we will not be able to make an accurate prediction. Instead, we aim to estimate the eviction time of cold objects that have not been accessed recently. Since eviction policies will likely evict these objects earliest, it will yield a more stable "minimum" eviction time. This approach allows handling hot objects as well, since we can make voluntary evict/expire predictions on each access. As long as the object is reaccessed, these predictions don't matter, but once the object is no longer accessed (becomes cold), then the predicted eviction time matters.

One way to estimate the eviction time for cold objects would be to measure the real time from last access of an object to its eviction. This approach works well for LRU but not for FIFO-like policies in which an access does not affect the position of an object in the eviction queue. In the FIFO case, we would be measuring the eviction time from the object's position in the queue to the tail of the queue, which would be a poor estimate of the eviction time for newly cached objects. We assume that a newly cached object that is not accessed again will traverse the entire eviction queue (e.g., from the head to the tail of the queue). Thus, on every eviction, we measure the eviction time of the object as the real time that the object has taken to traverse the eviction queue from the head of the queue until its eviction. This requires storing the last time when the object was inserted at the head of the queue, e.g., last access time for LRU in the `evictMetadata` field shown in Figure 3.3.

Psyche tracks the distribution of eviction times in a histogram. It updates this histogram on each eviction, and uses it to derive eviction time statistics such as the median eviction time, which we use as the *predicted eviction time*. By default, we use the median since it is less affected by outliers compared to the mean. However, unlike the mean, the median is difficult to calculate efficiently from a stream of data, and so we update the median periodically.

We choose an *update period* between 15 minutes to 1 hour since prior work on dynamically changing cache eviction policies [19] shows that changing policies once per hour provides a good balance between adaptability to workload changes and stability. After calculating the median, we record a count of eviction times that are above (call this count A) and that are below (call this count B) the current median. At the beginning of the next update period, if the A/B ratio lies within the $[49\% - 51\%]$ range, we consider the workload to be stable and do not calculate the median. Otherwise, we recalculate the median by scanning the histogram. The scan has negligible cost since it is done relatively infrequently.

When Psyche makes a prediction, as described in Subsection 3.2.1, it compares the TTL of the object to the predicted eviction time. If the TTL is smaller, then the object is placed in the expiry queue, since it is more likely to expire before it is evicted. Otherwise, it is placed in the eviction

queue.

3.2.3 Initial Estimation of Eviction Time

Psyche generates its eviction time histogram based on the evictions that have happened in the past. With a cold cache, this histogram isn't populated yet, and so we need some method for bootstrapping the predicted eviction time. One option for accurately estimating the eviction time is to profile a similar workload, but this requires per-workload profiling. A second option is to use an underestimate of the eviction time, for example, starting at 0. In this case, all objects are initially placed in the eviction queue. The drawback of this option is that expired objects will remain in the eviction queue after their expiry and fill the cache until they are discarded on reaccess or evicted once the cache is full.

The final option, and the one we use, is to overestimate the eviction time. In this case, all objects are initially placed in the expiry queue, which ensures that expired objects are discarded correctly. However, until we learn the accurate estimate of the eviction time, it is possible that all objects will be in the expiry queue and the cache reaches its capacity. Thus, we introduce a secondary eviction policy called *TTL eviction* that operates when the eviction queue is empty. This policy evicts objects from the expiry queue in the order of soonest expiring object. This is an eviction since we are evicting unexpired objects and these evictions also update the eviction time histogram. The eviction time is defined as the time from when the object would have been placed at the head of the eviction queue until its eviction (e.g., for LRU, this would be the last reaccess time until TTL eviction). We use this policy since objects at the end of their TTL are accessed less frequently [29].

The advantage of our approach is that by intentionally biasing toward placing objects in the expiry queue, we are able to handle expired objects effectively while the cache is filling up. The main drawback of our approach is that when the cache starts evicting objects the miss rate may initially be high since the evictions are performed entirely based on TTLs, similar to the TTL-only policy described in Section 2.2. The TTLs of these objects may not be close to their expiry and would therefore be uncorrelated with the access patterns.

However, the TTL eviction policy will allow Psyche to obtain an initial estimate of the eviction time because an object evicted by TTL eviction often has a shorter eviction time than that of the same object evicted by the regular eviction policy. For example, LRU evicts the least recently used object, or the object with the longest eviction time (since last access, as defined in Section 3.2.2). Thus an object evicted by any other policy, including TTL eviction, will have a shorter eviction time, which will shorten the predicted eviction time and prompt more objects to be placed in the eviction queue. As more evictions occur, the predicted eviction time will eventually converge to its correct value.

3.2.4 Handling Mispredictions

A misprediction occurs when Psyche predicts that an object will be removed by one policy, but is removed by the other policy. There are two types of mispredictions, *delayed expiry* and *delayed eviction*, and they both arise when an object's eviction time differs from the predicted eviction time by more than the time buffer shown in Figure 3.4, which is known at prediction time. However, the true eviction time (i.e., the real time when the object is evicted) and therefore the prediction error

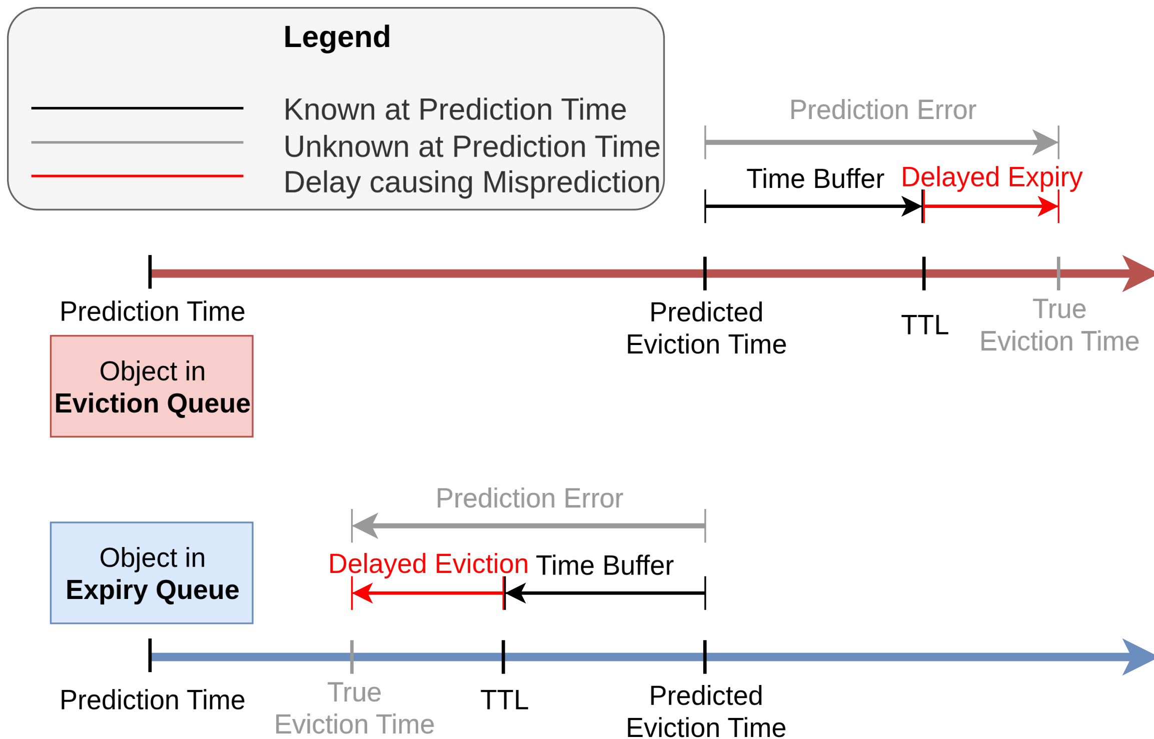

Figure 3.4: Mispredictions in Psyche.

are unknown until the object is evicted.

The top of Figure 3.4 shows that a delayed expiry occurs when an object stays in the eviction queue beyond its expiry. At prediction time, the TTL of this object was less than the predicted eviction time. However, since the object should be evicted, its true eviction time must be less than its predicted eviction time. This situation is caused, for example, by an increase in the number of incoming objects, which speeds up evictions and lowers object eviction times. When this happens, such expired objects will unnecessarily consume memory. PSYCHE removes expired objects in two ways. If they are accessed, they will be discarded. However, expired objects are unlikely to be accessed, and so they are eventually evicted. Since the eviction time of these objects are higher than the predicted eviction time, and we estimate eviction time based on median eviction times, the predicted eviction time will eventually increase and thus reduce these mispredictions.

The bottom of Figure 3.4 shows that a delayed eviction occurs when an object stays in the expiry queue beyond the time when it would have be evicted, had the object been in the eviction queue. At prediction time, the TTL of this object was less than the predicted eviction time. However, since the object should be evicted, its true eviction time must be less than its predicted eviction time. This situation is caused, for example, by an increase in the number of incoming objects, which speeds up evictions and lowers object eviction times. When this happens, the miss rate will potentially rise. For example, LRU/PSYCHE may place a recently accessed object in the eviction queue due to a long TTL, and this object may be evicted earlier than another less recently accessed object placed in the expiry queue due to a short TTL. A second issue with delayed evictions is that PSYCHE could run out of objects in the eviction queue, but the cache reaches its capacity. As described in Subsection 3.2.3, TTL evictions handle this case. Whether a recently accessed object is evicted or

TTL evictions occur, the eviction time of these objects is lower than the predicted eviction time, and so the predicted eviction time will eventually decrease and reduce these mispredictions.

The two cases described above show that PSYCHE eventually learns the true eviction time on a stable workload. Our predictor performs well when the time buffer between an object's TTL and predicted eviction time is large (i.e., the predicted eviction time is much larger or smaller than the TTL). In this case, the object is likely to be placed in the correct queue and our caching policy will accurately simulate the true caching policy (e.g., LRU/Proactive-TTL). When the object's TTL is close to the predicted eviction time (the time buffer in Figure 3.4 is small), mispredictions will occur since our median-based estimate will lead to only 50% accuracy. However, given that the TTL and the eviction time are similar, even when the object is removed (evicted or discarded) from the cache for the wrong reason, it will be removed at approximately the correct time, and so PSYCHE will still perform reasonably well.

3.2.5 Handling Changing Workloads

Real-life workloads experience changes in the frequency of requests and the working set of objects over time. For example, some caches experience increased traffic with the daily ebb and flow of day versus night, while others may experience bursts of traffic during major news events, such as election results or natural disasters. These bursts of activity require adapting various caching strategies, such as sizing the cache correctly with an MRC, or choosing the best eviction policy [19].

A changing workload affects PSYCHE when the object eviction times deviate significantly from our predicted eviction time. Recall from Subsection 3.2.2, we periodically recalculate the median using the eviction time histogram, which tracks the eviction times of all evictions in the past. To handle a changing workload, we prioritize more recent eviction times by decaying the frequencies of past eviction times in the histogram before recalculating the median. We use a scale factor of 0.5 to update the frequencies.

When the object eviction times change rapidly, then PSYCHE may experience many delayed expiries and evictions. Recall from Subsection 3.2.4, a delayed expiry occurs in the eviction queue when eviction times increase significantly, due to drop in the number of incoming objects. We describe how PSYCHE would handle this below, but we have not yet implemented this.

PSYCHE would keep track of the ratio of recently evicted objects that are expired. When this ratio reaches 1%, it would voluntarily predict by sampling 100 objects at a time from the eviction queue (starting from the LRU object) and discarding the expired objects, and continue until fewer than 1% of objects are expired. A workload change may also cause delayed evictions to occur in the expiry queue when the eviction times decrease significantly, due to an increase in the number of incoming objects. However, this case is less likely because object TTLs decrease over time. It would keep track of the ratio of recently expired objects that are evictable (e.g., with LRU, their last access times are older than the last access time of the most recently evicted object). When this ratio reaches 1%, it would voluntarily predict by sampling 100 objects at a time from the expiry queue (starting from soonest expiring object) and moving evictable objects to the tail of the eviction queue in no particular order relative to other evictable objects, and continue until fewer than 1% of objects are moved.

3.3 Psyche Eviction Policies

This section presents the Psyche implementations of the LRU and LFU eviction polices that support both evictions and expiries efficiently. We chose these policies because they are commonly used and supported by popular caches such as Redis [16]. For both policies, we describe when we make predictions.

3.3.1 LRU/Psyche

For LRU/Psyche, we make voluntary predictions when an object is reaccessed because a reaccess is a strong signal that our old predicted eviction time is likely incorrect. Removing or adding an object is an $O(1)$ operation for the LRU queue and an $O(\log N)$ operation for the expiry queue. Typically, objects tend to move once from the LRU queue to the expiry queue. After that, since an object's TTL decreases over time, it is likely that its TTL remains shorter than the predicted eviction time as it ages. Since LRU evicts least recently used objects with the longest eviction time (since last access), any mispredictions due to delayed evictions will shorten the predicted eviction time, as explained in Subsection 3.2.4. This will lead to more objects being correctly placed in the eviction queue, which should also improve the accuracy of the predicted eviction time.

3.3.2 LFU-f/Psyche

We begin by describing how Psyche can be implemented for an exact version of LFU. Then, we introduce a more memory efficient approximate version of LFU called LFU-f. LFU works by evicting the least frequently accessed object. Upon ties, it will evict the least recently used among the least frequently used. For this reason, LFU can be modeled accurately as a collection of logical LRU (or FIFO[1]) queues, each associated with a specific access frequency and its own predicted eviction time. However, this approach requires tracking logical LRU queues for each frequency and an eviction time histogram for every evicted frequency.

Luckily, we can use the properties of LFU to approximate it well. LFU evicts lower frequency objects before higher frequency objects and so the higher frequency objects tend to be old and more likely to expire. Thus, low frequency objects may either be evicted or expired, while higher frequency objects will probably expire. We can thus approximate LFU/Psyche by tracking the logical LRU queues and eviction time histograms for the lowest f frequencies. Upon an insertion to a queue, we make a prediction of whether the object will expire before being evicted. If so, then it is moved to a single expiry queue. However, any object with a frequency greater than f is always placed in the expiry queue, regardless of its TTL. We call this approximation LFU-f/Psyche. Similar to LRU/Psyche, we learn the distribution of eviction times for each logical LRU queue. For LRU caches that have experienced no evictions due to LFU, we use TTL evictions to populate their eviction time histogram.

For a particular workload and cache size, LFU-f/Psyche behaves identically to LFU/Psyche when no objects with a frequency greater than f are evicted. This could happen when the cache size is large, or TTLs are sufficiently short, or there are few objects with frequency greater than f. Even

[1] Accessing an object causes it to be moved to a different logical queue, so the distinction is moot. We say LRU because we have previously described LRU/Psyche.

when objects with frequencies greater than f are evicted from LFU/Psyche, these tend to be rarer than evictions of objects with lower frequencies, making LFU-f/Psyche a good approximation.

By default, our implementation uses LFU-1/Psyche because many objects in real-world workloads are accessed only one time [30]. Our evaluation shows that for most workloads and cache sizes, LFU-1/Psyche simulates LFU/Proactive-TTL accurately. Our implementation of LFU-1/Psyche is nearly identical to LRU/Psyche, except when an object is reaccessed, it is immediately placed in the expiry queue regardless of its TTL. Thus, the eviction queue tends to be shorter than for LRU/Psyche and it is more common for the eviction queue to be empty. If this is the case, we use TTL-evictions, similar to LRU/Psyche.

3.4 Discussion

Choice of Predictor Psyche is a binary classifier that tries to predict whether to place an object in the eviction queue or the expiry queue. A more flexible classifier could account for uncertainty in the predicted eviction time by using two (low and high) eviction time thresholds. Objects with TTLs below the low threshold are placed in the expiry queue and objects with TTLs above the high threshold are placed in the eviction queue. Objects with TTLs between the low and high thresholds are placed in both queues. However, the flexibility of an object being in either one or both queues increases complexity and reduces memory savings. Our evaluation in Chapter 4 shows that our binary classifier provides good results without needing to resort to placing objects in both queues.

A predictor could also avoid placing an object in either queue if it is sure that the object will be accessed again before it should expire or be evicted. However, in this case, if the prediction is wrong, then the object may stay in the cache for a long time and would need to be removed via some scanning process, and so we do not think this option is viable.

Multi-Queue Eviction Policies Currently, we have implemented the LRU and LFU eviction policies using Psyche. Other single-queue eviction policies such as FIFO and Second-Chance should be relatively easy to implement using Psyche. However, multi-queue eviction policies such as 2Q and S3-FIFO will require more work. Similar to the way we approximate LFU with LFU-f, with multiple logical LRU queues and eviction histograms, we may be able to approximate these policies. Each queue within the multi-queue policy may have its own expiry queue or share it with the other queues, as in LFU-f. We leave these extensions for future work.

Chapter 4

Evaluation

We evaluate the Psyche implementations of the LRU and the LFU eviction policies by comparing them against the Proactive-TTL and Periodic-TTL policies. We show that the Psyche policies provide significant memory savings over the Proactive-TTL policies due to the reduced TTL metadata overhead, while having comparable miss ratios. Then, we also show that the Psyche's expiry policy searches through many fewer objects compared to the Periodic-TTL policies used by Memcached and CacheLib, but are able to find more expired objects, thus lowering memory usage. We exclude Redis from our evaluation of Periodic-TTL policies because of difficulties accurately simulating it with SHARDS sampling.

Section 4.1 describes the dataset we use for our experiments and Section 4.2 describes our experimental method. Section 4.3 presents our evaluation results and Section 4.4 evaluates our design choices in more detail.

4.1 Experimental Dataset

We use the Twitter traces, representing the cache operations of Twitter's (now X) 54 largest caches, for our experiments because they are both large-scale and Twitter mandated the use of TTLs in their caches. The data is typically collected over the course of 1 week [28]. The Twitter traces use look-aside cache semantics, whereby the cache client is responsible for populating the cache from the back-end storage system.

We ran our experiments using a subset of Twitter traces recommended by Yang et al. [28], consisting of traces collected from Clusters #6, #7, #11, #12, #15, #17, #18, #19, #22, #24, #25, #29, #31, #37, #44, #45, and #52. Table 4.1 categories these traces.[1] We remove the traces shown in *italics*. Clusters #12, #15, #31, and #37 have less than a million unique objects (since we only consider read requests, as described later) and Cluster #44 has no expiring objects, which leaves us with 12 workloads.

We use the publicly-available preprocessed Twitter traces provided by Sultan et al. [21]. These traces use the GET requests from the workloads, as in previous studies [26, 4]. The SET requests are separated because they were captured when running a specific (undisclosed) cache size and so their number and placement would be different for different cache sizes. Instead, for GET requests

[1] Clusters #18 and #52 are listed twice in this table because they fall into multiple categories.

Evaluation Purpose	Clusters
Miss Ratio Related	High Reuse: #17, #18 Low Reuse: #24, #29, *#44*, #45, #52
Write-Heavy	*#12*, *#15*, *#31*, *#37*
TTL Related	Both small and large TTLs: #11, #22, #25, #52 Only small TTLs: #6, #7, #18, #19

Table 4.1: Recommended Twitter traces, as per Yang et al.

that cause a miss, we issue a SET request, as in a look-aside cache. When we issue a SET request, we must attach a TTL, but Twitter's GET requests do not include TTL information. Therefore, we infer the TTL for our SET requests from Twitter's SET requests corresponding to the same key. Thus, we ignore GET requests without any SET requests for the same key in the trace.

4.2 Experimental Method

We simulate caching policies instead of running the caches in real time, which would significantly slow the experiments. The timestamps in the Twitter trace have 1 second granularity and so our simulation uses this granularity to handle all accesses and expirations.

We measure three metrics, *miss ratio*, *memory usage* and the number of expiry-related *probing accesses*. Memory usage measures the sum of the sizes of all objects (keys and values) and their metadata. Both expired objects and TTL-related metadata increase memory usage. Object accesses measures the number of objects searched while expiring objects, which approximates compute costs for a memory bound cache.

4.2.1 Simulating Periodic-TTL Policies

We simulate the Periodic-TTL policies in Memcached and CacheLib. Next, we briefly describe them and how we simulate their policies.

Memcached is a widely-adopted, open-source in-memory cache. It groups objects by approximate size in slab classes. When the cache runs out of memory, it will evict an object from the same slab class as the incoming object. We simulate the Periodic-TTL policy used by Memcached by copying its expiry policy into our simulator. Memcached aims to have no more than roughly 1% of expired objects in its cache. It removes expired objects using a combination of lazy TTL eviction and an LRU crawler that periodically scans each Memcached slab's LRU eviction queues for expired objects [10]. The initial and the minimum time between scans is one minute and the maximum time is one hour. The crawler decides when to scan next by generating a histogram of TTLs. It aims to schedule the next run when 1% of the objects expire. If 1% of objects expire before the next scan, then the next scan is scheduled a minute sooner, otherwise the scan is scheduled a minute later.

CacheLib is a library for creating caches on both DRAM and flash developed at Facebook. CacheLib's expiry policy removes expired objects by periodically scanning the entire cache every 10 seconds [12]. This is an aggressive policy that ensures that objects are discarded within 10 seconds of expiry.

For Memcached and CacheLib, instead of scanning objects, we speed up the simulation by using

Key	Timestamp [sec]	TTL [sec]	Expiration Time [sec]
A	1	4	5
B	2	100	102
C	3	100	103
A	4	4	8
A	5	4	9

Table 4.2: An example trace where LRU/TTL violates the inclusion principle.

Timestamp [sec]	Cache of Size 2		Cache of Size 3	
	LRU Stack	Hit/Miss	LRU Stack	Hit/Miss
1	A(exp@5)	Miss (Cold)	A(exp@5)	Miss (Cold)
2	B(@102),A(@5)	Miss (Cold)	B(@102),A(@5)	Miss (Cold)
3	C(@103),B(@102)	Miss (Cold)	C(@103),B(@102),A(@5)	Miss (Cold)
4	A(@8),C(@103)	Miss (LRU-Ev)	A(@5),C(@103),B(@102)	Hit
5	A(@8),C(@103)	Hit	A(@8),C(@103),B(@102)	Miss (TTL-Exp)

Table 4.3: An example of two caches where LRU/TTL violates the inclusion principle.

an expiry queue to discard objects, and use the number of objects in the cache to calculate the number of probing accesses. This makes their expiry policy appear more effective because it expires objects faster and thus reduces memory usage more rapidly.

4.2.2 Miniature Simulations

We use Miniature Simulations (Minisim) [23] in our simulations to generate miss ratio curves for different cache sizes. Minisim uses SHARDS sampling to reduce the number of unique objects in the cache, which reduces memory requirements and helps speed up our experiments [24]. We use a sampling rate of 0.001 and make the SHARDS adjustment for the miss ratio [24]. Due to Minisim's sampling, we scale up the number of objects, and therefore memory usage and the number of probing accesses, by a factor of 1000. For Psyche, we scale the number of probing accesses by an additional factor of $O(\log M)$, where M is the total number of objects in the cache after scaling, since Psyche uses a balanced binary search tree for expiring objects.

Prior work claims that LRU/TTL is a stack algorithm and so its MRC can be generated efficiently in one pass [21]. We demonstrate that it violates the inclusion principle in the trace shown in Table 4.2. A specific example of how this trace violates the inclusion principle is shown in Table 4.3. In this example, object A is evicted from the smaller cache but then reinstalled with a refreshed TTL. However, in a larger cache, object A's TTL is not refreshed, leading to object A being present in the smaller cache but not in the larger cache at the end of the trace. Thus, LRU/TTL is not a stack algorithm, and hence we use Minisim, which can generate miss ratio curves for non-stack algorithms.

4.3 Evaluation Results

In this section, we first present results for miss ratio, memory usage, and the number of objects accessed while expiring objects.

4.3.1 Miss Ratio

Figure 4.1 shows the miss ratio curves for the TTL-only, LRU/Lazy-TTL, LRU/Proactive-TTL, and LRU/Psyche policies. This figure shows that the TTL-only policy consistently yields a worse miss ratio than the other algorithms. LRU/Psyche achieves performance comparable to LRU/Proactive-TTL. Counter-intuitively, the miss ratio curves are similar for the Proactive-TTL and the Lazy-TTL policies. With Lazy-TTL, we expect that expired objects may displace some lesser used but unexpired objects that are accessed in the future. However, in the Twitter traces, there is a strong correlation between object age and last access, so expired objects tend to exist near the eviction tail of the LRU, and so LRU tends to evict expired objects. Even so, Lazy-TTL has high memory usage, as shown in Subsection 4.3.2.

Figure 4.2 shows the miss ratio curves for the LFU/Proactive-TTL, LFU-1/Proactive-TTL and LFU-1/Psyche policies. This figure shows that LFU-1/Psyche mostly matches the LFU/Proactive-TTL policy. The most significant differences are in Clusters #24 and #52. The plot of LFU-1/Proactive-TTL helps show how much of the difference between LFU-1/Psyche's MRC and LFU/Proactive-TTL's MRC is due to the one queue approximation. Subsection 4.4.3 shows that Psyche can match the miss ratio of LFU/Proactive-TTL closely when it uses more queues to track more frequently accessed objects.

4.3.2 Memory Usage

This section presents the results of four memory usage experiments showing that Psyche outperforms all Lazy-TTL, Proactive-TTL and Periodic-TTL algorithms.

Lazy-TTL

Figure 4.3 shows memory usage for the LRU/Proactive-TTL and the LRU/Lazy-TTL policies for the Twitter Cluster #19 workload at three different cache sizes. With Lazy-TTL, memory usage quickly saturates since expired objects are not discarded, while the Proactive-TTL policy has lower memory utilization since expired objects are discarded immediately. This difference becomes more significant for larger caches. Since caches using the Lazy-TTL expiry policy fill the entire cache size, it provides limited insight into how the cache should be sized. Similar results hold for all the Twitter workloads and cache sizes.

Proactive-TTL

Compared to Proactive-TTL, Psyche saves memory usage by reducing per-object metadata. However, delayed expiries increase its memory usage. Figure 4.4 shows the amount of memory LRU/Psyche saves over LRU/Proactive-TTL with a small 1 GiB cache, a medium 4 GiB cache, and a larger 8 GiB cache (which is greater than the working set size for all but two workloads, clusters #24 and #52). These experiments do not decay the eviction time histogram.

The memory savings are identical for all cache sizes when the maximum memory usage is under 1 GiB, such as for clusters #6, #18, #22, and #25. Similarly, the memory savings are identical for the 4 GiB and 8 GiB cache sizes when the maximum memory usage is under 4 GiB, such as for clusters #7, #11, #17, #29, and #45. Clusters #19, #24, and #52 have higher maximum memory usage.

This figure shows that for Cluster #52, PSYCHE saves up to 700 MiB of memory compared to LRU/Proactive-TTL for a cache size of 8 GiB. The average object size is only a couple of hundred bytes in this workload, so the 16-byte saving per object represents roughly 8% of the object size, which matches with the savings.

LRU/PSYCHE provides memory savings compared to LRU/Proactive-TTL for all workloads except Cluster #19. Cluster #19 exhibits significant workload changes, with large, rapid memory usage fluctuations between 2 GiB and 5 GiB, as shown by the memory usage of LRU/Proactive-TTL in Figure 4.3. When the cache size is 1 GiB, memory usage reaches the cache size, and the constant evictions provide a good prediction of the eviction times, which is why PSYCHE has lower metadata overhead. When the cache size is 8 GiB, the dataset fits in the cache, so all objects are placed in the expiry queue and PSYCHE again has lower metadata overhead. However, when the cache size is 4 GiB, memory usage saturate twice, as shown in Figure 4.3. When the true memory usage (i.e., the memory usage of the LRU/Proactive-TTL cache) starts to decrease, evictions stop or slow down, raising eviction times, which causes delayed expirations and increased memory usage, as described in Subsection 3.2.4. Subsection 4.4.2 shows that decaying the histogram mitigates this problem.

Figure 4.5 shows similar results for LFU-1/PSYCHE compared to LFU/Proactive-TTL.

Periodic-TTL

We show that PSYCHE finds and discards more expired objects than the Periodic-TTL policies. In this experiment, we size our cache sufficiently large to avoid any evictions so that we can focus solely on expiries. Figure 4.6 shows the ratio of maximum memory usages within every one hour time interval for Memcached and CacheLib compared to PSYCHE. PSYCHE discards expired objects immediately (once per second) from its expiry queue. As expected Memcached uses roughly 1% extra memory over PSYCHE since it aims to keep about 1% expired objects. Memcached has higher memory usage for cluster #6 because this is a small workload with few objects per slab class that are missed by the granularity of Memcached's 1% expiry policy. CacheLib uses little extra memory because of its frequent scans for expired objects. However, this frequent scanning increases the number of probes dramatically, as discussed below.

To validate our findings, we look at Figure 4.7, which shows the number of objects searched (in italic text) and discarded (in bold text) by Memcached, CacheLib, and PSYCHE for each Twitter workload. The more expired objects that a policy discards, the better since this means that it is wasting less memory on expired objects. The geometric mean ratios of discarded objects of Memcached and CacheLib compared to PSYCHE are 99.9% and 99.999%, respectively. This shows that Periodic-TTL approaches, but does not attain, PSYCHE's ability to discard expired objects in large caches.

4.3.3 Probing Accesses

Figure 4.7 shows the number of the expiry-related probing accesses in Memcached, CacheLib, and PSYCHE. The solid colors and the corresponding numbers (in bold text) show the number of expired objects that were discarded over the entire experiment. These numbers are similar for all the policies. The light colors and the corresponding numbers on top of each bar (in italic text) show the number of probing accesses. PSYCHE's number of probing accesses is related to the number of expired objects, while Memcached and CacheLib's is related to the number of objects in the cache. We observe that PSYCHE needs to search a geometric mean of 8.9× fewer objects than Memcached and 454× fewer objects than CacheLib. PSYCHE performs particularly well on traces with many objects and long TTLs.

4.4 Sensitivity Analysis

This section analyzes the effect of changing the PSYCHE parameters.

4.4.1 Alternate Eviction Time Statistics

Figure 4.8 shows the miss ratio curves and memory savings for a 4 GiB cache for Cluster #19 when we use the median, mean, 25th-percentile, and 75th-percentile for the predicted eviction time. All the MRCs are roughly the same, but we get significantly different behavior for memory usage. The mean and 75th-percentile caches' memory savings dip lower than the median around 100 hours, but avoid the second dip at 150 hours. This suggests that a bias toward placing objects in the expiry queue may yield better memory savings for no miss ratio penalty. However, a scanning strategy will likely be a more effective way for handling the delayed expirations, as discussed above.

Figure 4.9 shows the predicted eviction times over time with a 1 GiB cache and a 4 GiB cache for Cluster #19. The eviction times in the 1 GiB cache tend to decrease over time before stabilizing because evictions take a while to get started at the beginning, but once they begin there are continual evictions because the cache is always full. However, the 4 GiB cache only has sporadic evictions, due to the expiries that occur in this larger cache size. The similarity between the behaviours of the mean and 75th-percentile caches explains the similarity in behaviour between these two caches.

4.4.2 Dynamic Workloads

For Cluster #19, PSYCHE is unable to discard expired objects as efficiently as LRU/Proactive-TTL with a 4 GiB cache. This is because this workload experiences rapid swings from expiry-only to evict-only. Figure 4.10 shows the impact of updating the predictions at 15 minutes and 1 hour, and decaying the eviction time histogram at those times. We mitigate the impact of delayed expirations by decaying the histogram more frequently. PSYCHE will need to incorporate the scanning strategy discussed in Subsection 3.2.5 to reduce memory usage further. Unlike Memcached's scanning strategy that is based on TTLs, our strategy is deployed on workload changes. For Cluster #19, we will need just two scans when the memory savings drop rapidly.

4.4.3 LFU Approximation

Figure 4.2 shows that for most workloads, LFU-1/Psyche closely matches LFU/Proactive-TTL, which matches our intuition that the LFU eviction policy should mostly evict objects with an access frequency of 1 since they comprise a large portion of objects and are evicted first [30].

However, Cluster #52 is a conspicuous counter-example, where LFU-1/Psyche deviates from LFU/Proactive-TTL. Figure 4.11 shows the miss ratio for the Psyche implementations of LFU-1, LFU-8 and LFU-512. As expected, as f increases, we match LFU/Proactive-TTL. However, the drawback to selecting a large f is the large number of histograms required.

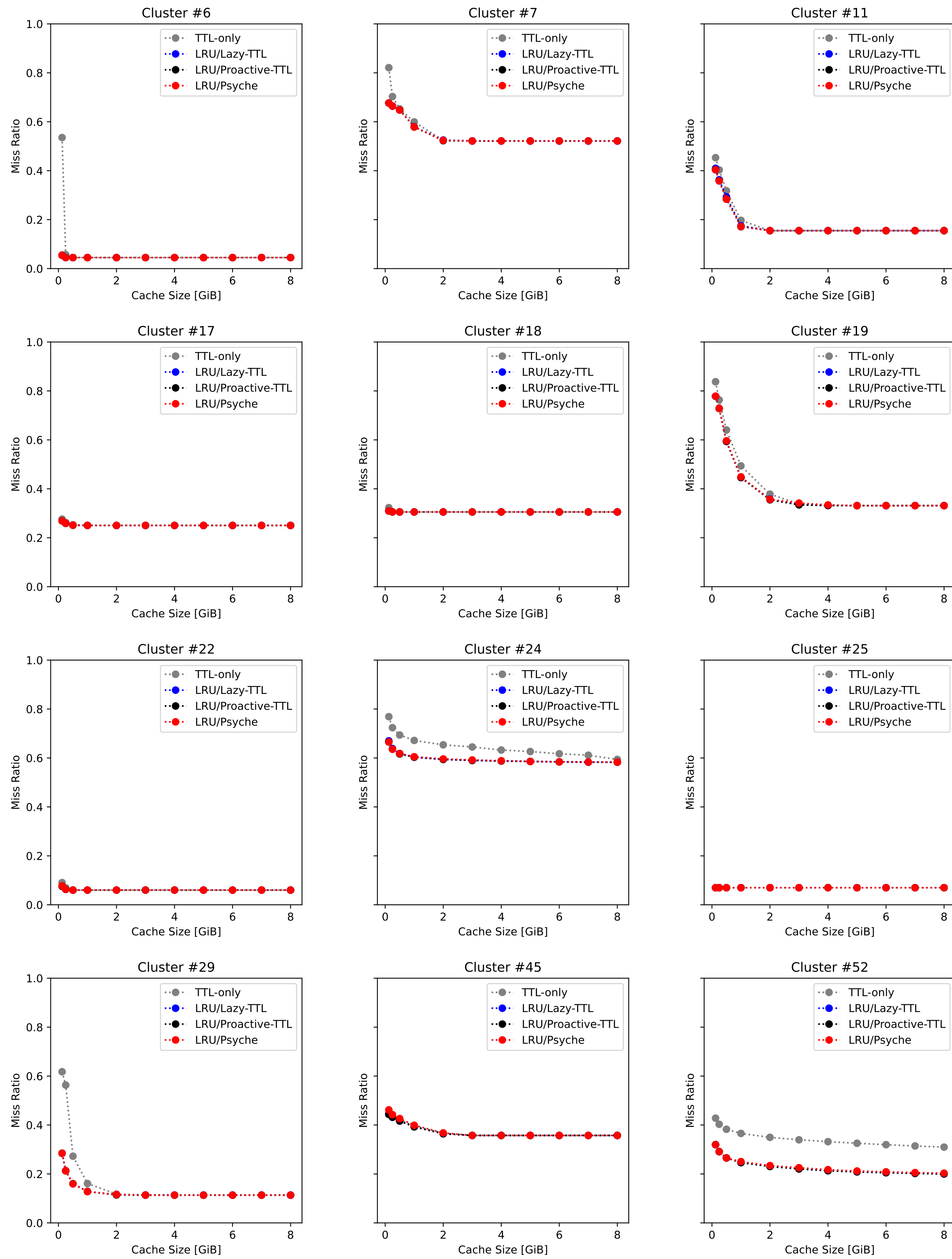

Figure 4.1: Miss ratio curves for LRU/Proactive-TTL, LRU/Psyche and TTL-only policies.

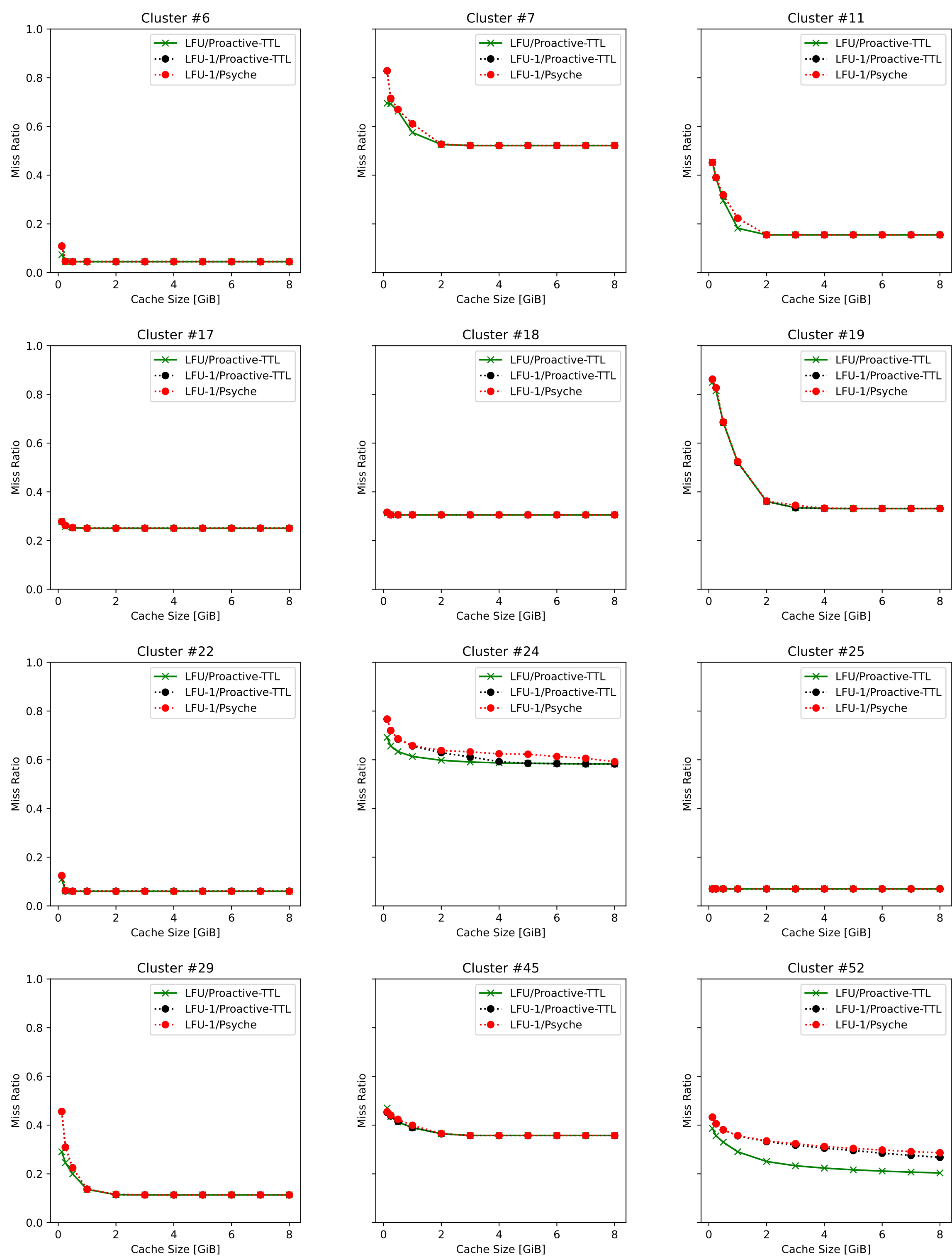

Figure 4.2: Miss ratio curves for LFU/Proactive-TTL, LFU-1/Proactive-TTL, and LFU-1/PSYCHE policies.

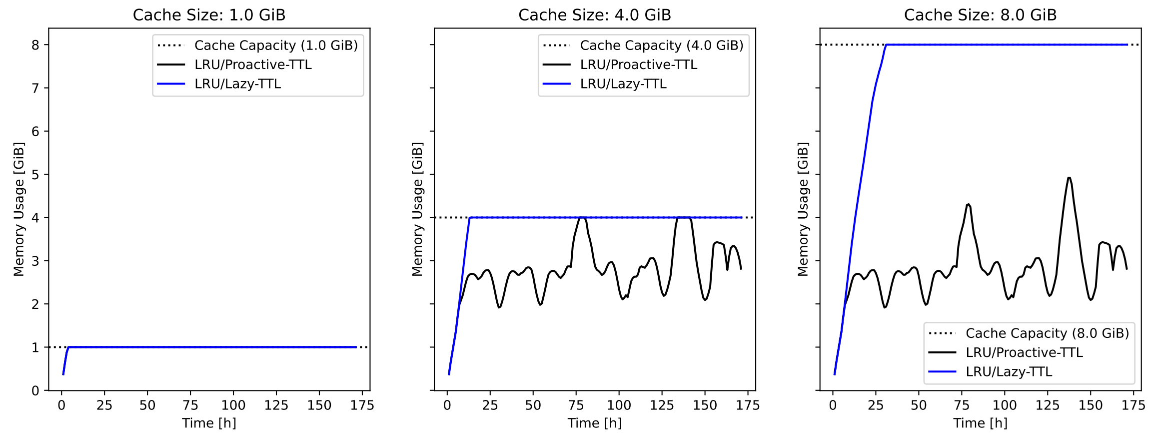

Figure 4.3: Memory usage for the LRU/Proactive-TTL and the LRU/Lazy-TTL caching policies for the Twitter Cluster #19 workload.

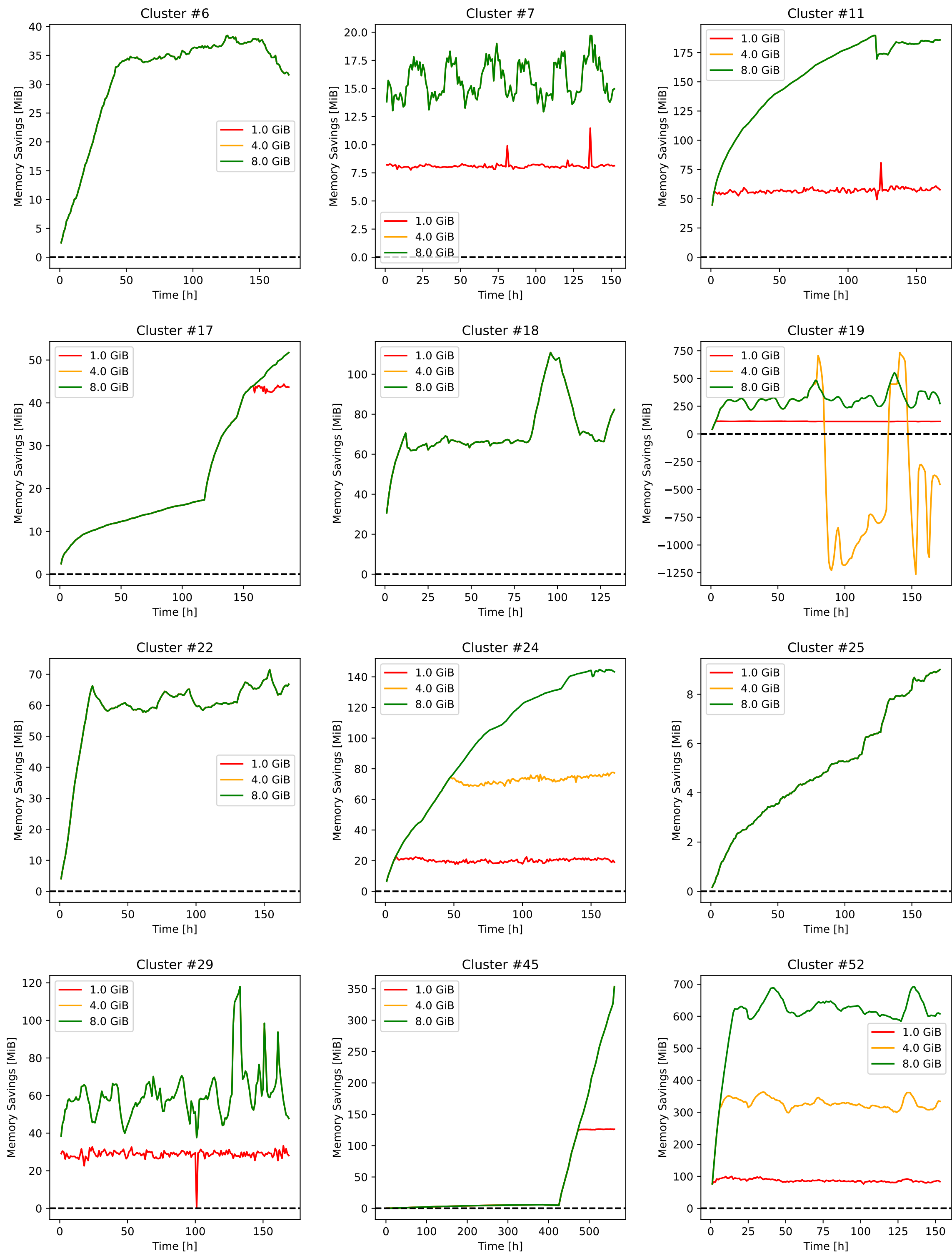

Figure 4.4: LRU/Psyche's memory savings over LRU/Proactive-TTL.

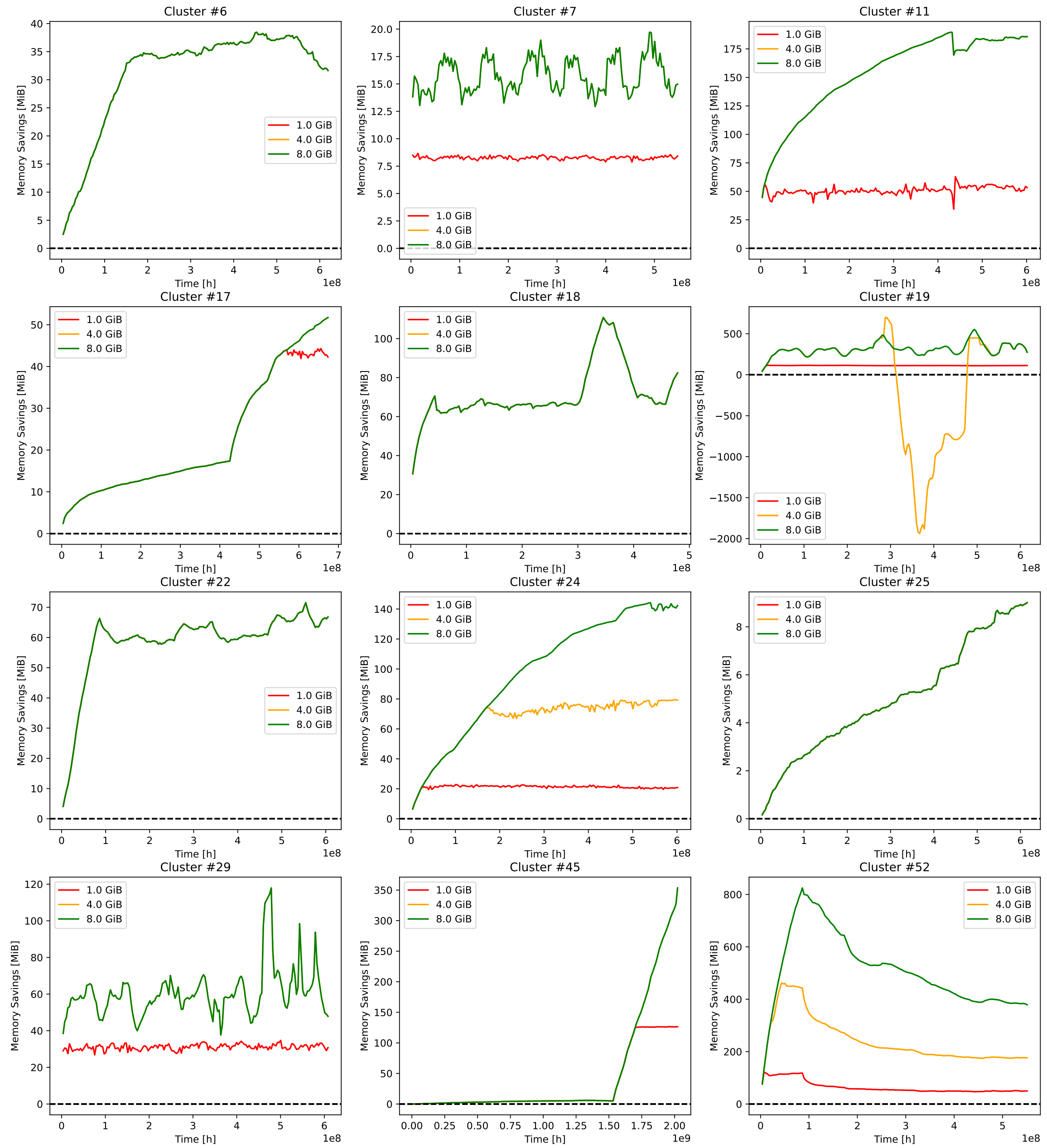

Figure 4.5: LFU-1/Psyche's memory savings over accurate LFU/Proactive-TTL.

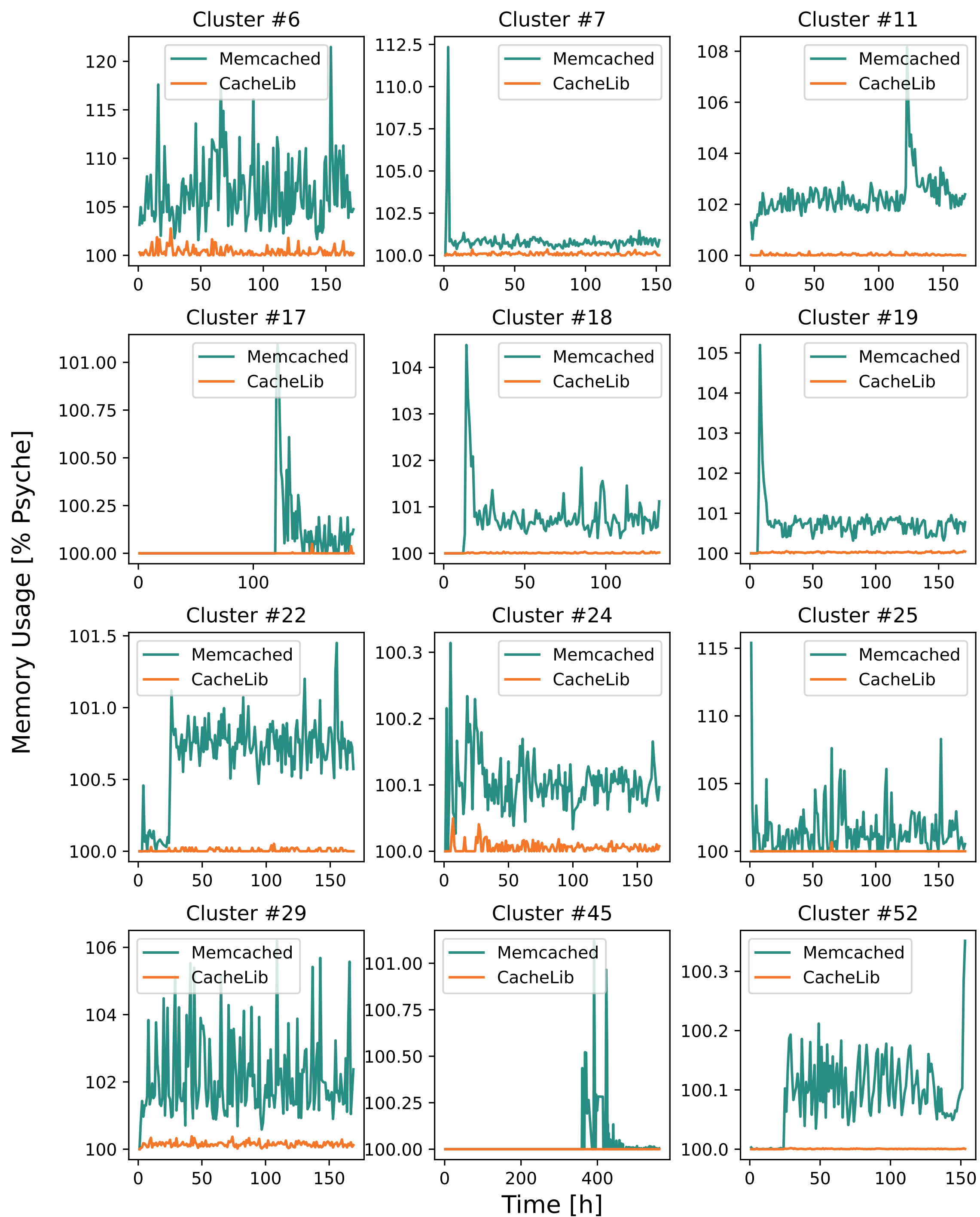

Figure 4.6: Memory usage of Periodic-TTL policy compared to Psyche.

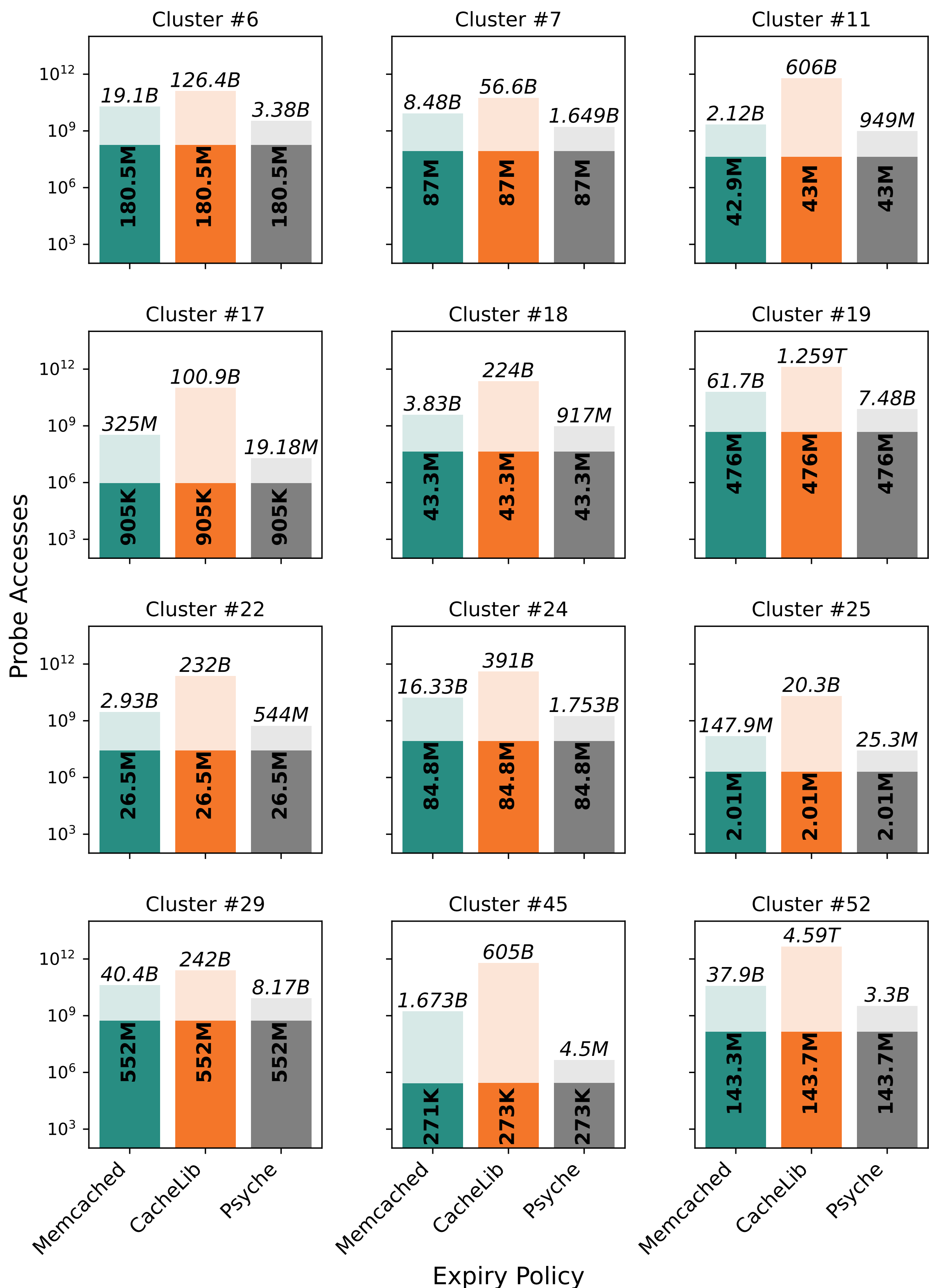

Figure 4.7: Object accesses by the expiry policy for Memcached, CacheLib and Psyche.

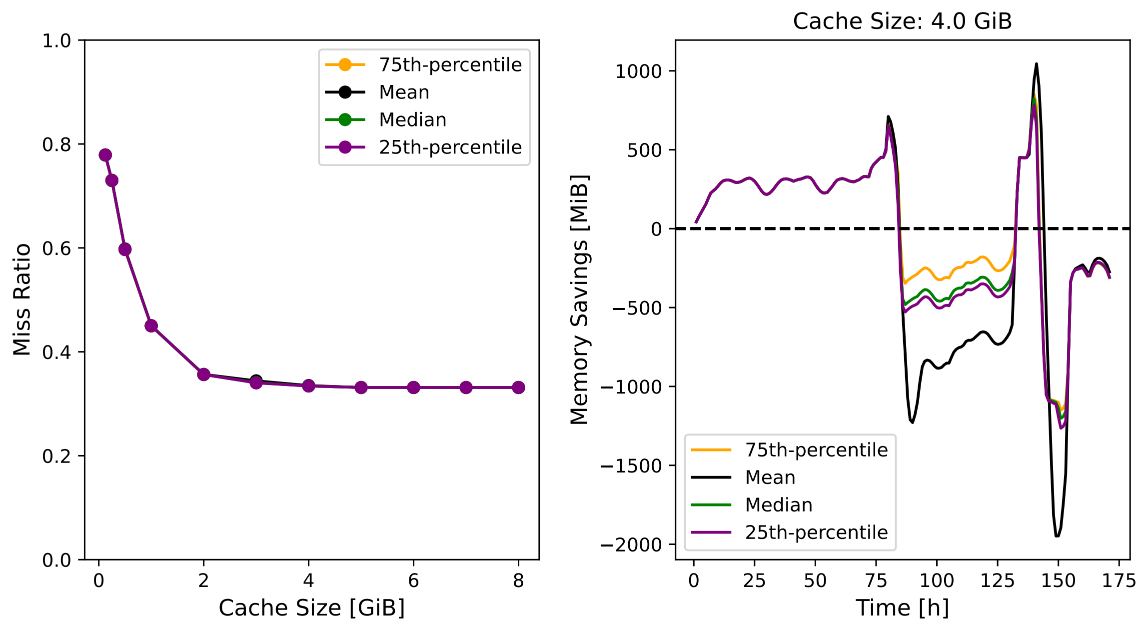

Figure 4.8: The miss ratio curves and memory savings for various predicted eviction times for a 4 GiB cache for the Cluster #19 workload.

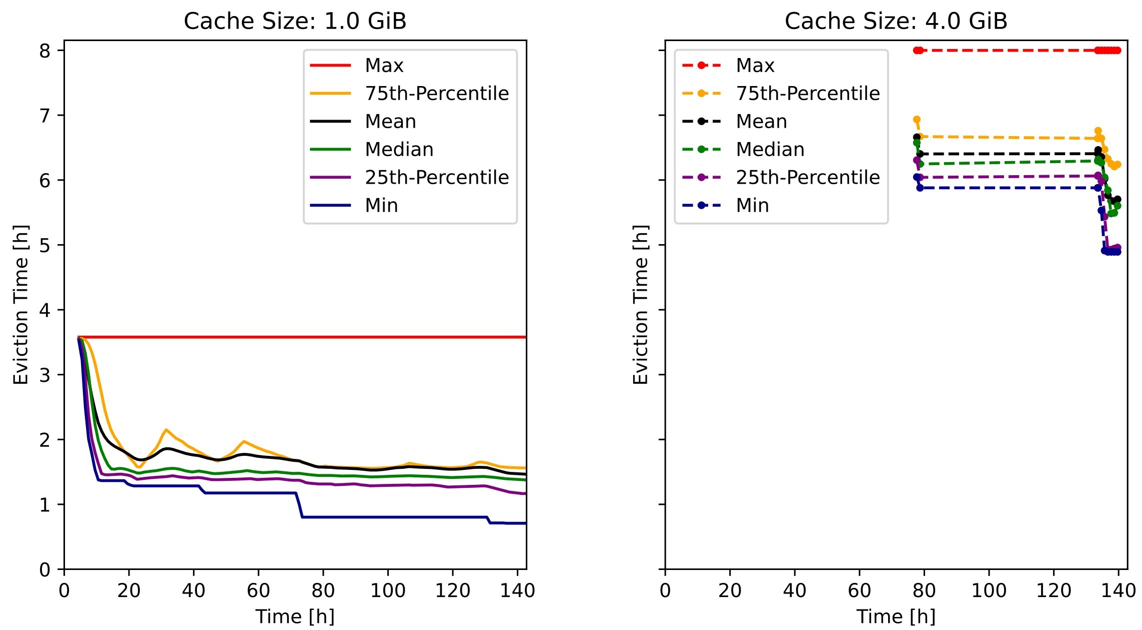

Figure 4.9: The changes in the eviction time for Cluster #19 using a 1 GiB cache and a 4 GiB cache.

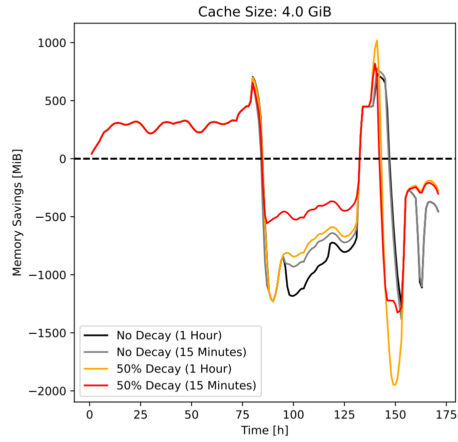

Figure 4.10: LRU/Psyche's memory savings compared to LRU/Proactive-TTL for varying levels of decay for Cluster #19.

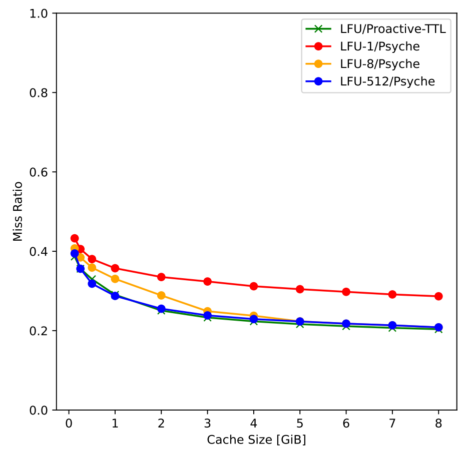

Figure 4.11: Increasing the number of logical LRU queues increases the LFU-approximation accuracy for Cluster #52.

Chapter 5

Conclusions

In-memory caches supporting both a size-limiting eviction policy and time-to-live attributes are ubiquitous in modern web-scale systems. However, these caches make a trade-off between reducing the need to search through the entire cache for expired objects and reducing memory usage [12, 8, 17].

We describe the design of Psyche, a caching mechanism that supports both cache evictions and time-to-live expiries efficiently. Psyche's novelty lies in predicting whether a cached object will first be evicted or expire, which enables supporting both a cache sizing eviction policy and time-to-live based expiry policy without the metadata overhead of supporting both policies or requiring scans to discard expired objects.

We evaluate Psyche on the recommended set of the Twitter traces [28] and show that it provides memory savings or operates more efficiently compared to state-of-the-art in-memory cache frameworks.

5.1 Future Work

In the following section, we discuss potential directions for future work in this area.

Probability vs Utility. Our method aims to maximize the probability of guessing correctly. However, the penalty of delayed expiries and delayed evictions is not necessarily symmetric. For example, an object kept beyond when it would have normally been evicted (e.g., due to LRU) may be reused and therefore yield a hit (although the LRU heuristic says this is unlikely). However, an object that has expired has no utility and will always yield a miss. Thus, we could bias our predictions towards the expiry queue.

Additional Eviction Policies. We implement Psyche for LRU and LFU. Extending this work for single queue eviction policies such as FIFO, Second-Chance, and Sieve should be relatively simple. However, multi-queue eviction policies, such as 2Q and S3-FIFO, are more challenging because objects move between queues depending on their access patterns and the relative number of objects in each queue. It may be possible to adapt the techniques used in our approximation of LFU to only simulate some queues.

Changing Workloads. Currently, we handle slowly changing workloads. Rapidly changing workloads will require scanning queues to remove expired and cold objects.

Efficient TTL Caches. Objects TTLs are monotonically decreasing and thus objects will tend to move from the eviction queue into the expiry queue and not the other way. For this reason, we can explore more memory-efficient packing of objects in the expiry queue [29], or combining PSYCHE'S eviction policies with SegCache's TTL cache.

Reducing Histogram Overhead. Tracking the expiry time of objects in a histogram creates some memory overhead. Future work may explore how to reduce this overhead by reducing the granularity of buckets while maintaining good accuracy.

Improving Initial Eviction Time Estimate. Currently, PSYCHE starts by predicting that all objects will expire. We may be able to better predict the initial eviction time by estimating the time at which a slowly growing cache will fill up. This estimation could be based on measurements, such as the rate of incoming new bytes, rate of existing objects' growth and shrinkage, rate of objects expiring by TTL, and the current cache size.

Unified Eviction Policies. Since we showed that LRU can be approximated with time-based heuristics, it is conceivable that we can place all objects in the expiry queue and eliminate the eviction queue altogether. Objects that are placed using the formerly-LRU policy will have refreshing TTLs, while objects placed due to their TTL will have non-refreshing TTLs. This would require having a good sense of the average LRU eviction time. A drawback is that the expiry queue must remain sorted and support arbitrary insertions and deletions, hence it will likely be $O(logN)$ unless approximations are made [22].

Application for DRAM and Flash Cache. Since objects tend to move from the capacity-based eviction queue, where the order is constantly being shuffled, to the TTL-based expiry queue, where the order is stable, future work can look at placing these queues in DRAM and Flash.

Supporting TTLs-with-Refresh. We can support TTLs-with-Refresh by using the updated TTL to predict whether an object will end up in the LRU or the expiry queue on each access. However, this would break the trend that objects tend to move from the capacity-based eviction queue to the TTL-based expiry queue.

Error Tolerance. Our experiments use a single threshold for making a prediction. However, if we define some uncertainty about our prediction, then we can make three types of predictions: LRU-only, TTL-only, and unsure. In the latter category, we simply place the object in both the LRU and expiry queues. This allows fine-grain tuning of the prediction accuracy.

Bibliography

[1] L. A. Belady, R. A. Nelson, and G. S. Shedler. "An anomaly in space-time characteristics of certain programs running in a paging machine". In: *Commun. ACM* 12.6 (June 1969), pp. 349–353. ISSN: 0001-0782. DOI: 10.1145/363011.363155. URL: https://doi.org/10.1145/363011.363155.

[2] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. "The CacheLib Caching Engine: Design and Experiences at Scale". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, Nov. 2020, pp. 753–768. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/berg.

[3] Gil Einziger, Roy Friedman, and Ben Manes. "TinyLFU: A Highly Efficient Cache Admission Policy". In: *ArXiv* (Dec. 2, 2015). URL: https://arxiv.org/pdf/1512.00727.

[4] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. "Kinetic Modeling of Data Eviction in Cache". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 351–364. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu.

[5] Intersoft Consulting. *General Data Protection Regulation (GDPR)*. May 23, 2018. URL: https://gdpr-info.eu/ (visited on 09/11/2025).

[6] Theodore Johnson and Dennis Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: *Proceedings of the 20th International Conference on Very Large Data Bases*. VLDB '94. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1994, pp. 439–450. ISBN: 1558601538.

[7] Alan "Dormando" Kasindorf. *New LRU*. Jan. 8, 2015. URL: https://github.com/memcached/memcached/blob/master/doc/new_lru.txt (visited on 07/07/2025).

[8] Alan "Dormando" Kasindorf. *Replacing the cache replacement algorithm in memcached*. Oct. 15, 2018. URL: https://memcached.org/blog/modern-lru/ (visited on 07/08/2025).

[9] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. "Evaluation techniques for storage hierarchies". In: *IBM Syst. J.* 9.2 (June 1970), pp. 78–117. ISSN: 0018-8670. DOI: 10.1147/sj.92.0078. URL: https://doi-org.myaccess.library.utoronto.ca/10.1147/sj.92.0078.

[10] Memcached. *Performance and Efficiency*. Oct. 15, 2019. URL: https://docs.memcached.org/serverguide/performance/#when-memory-is-reclaimed (visited on 07/07/2025).

[11] Meta Platforms, Inc. *Eviction Policy*. URL: https://cachelib.org/docs/Cache_Library_User_Guides/eviction_policy/ (visited on 09/02/2025).

[12] Meta Platforms, Inc. *TTL Reaper*. URL: https://cachelib.org/docs/Cache_Library_User_Guides/ttl_reaper/ (visited on 07/23/2025).

[13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. 2013, pp. 385–398.

[14] Frank Olken. "Efficient methods for calculating the success function of fixed-space replacement policies". Master of Science. Berkeley, California, USA: University of California, Berkeley, May 22, 1981. URL: https://doi.org/10.2172/6051879 (visited on 09/03/2025).

[15] Redis, Ltd. *Command key specifications*. URL: https://redis.io/docs/latest/develop/reference/key-specs/ (visited on 09/06/2025).

[16] Redis, Ltd. *Eviction Policies*. URL: https://redis.io/docs/latest/develop/reference/eviction/ (visited on 07/02/2025).

[17] Redis, Ltd. *What Are the Impacts of the Redis Expiration Algorithm?* Mar. 22, 2024. URL: https://redis.io/kb/doc/1fqjridk8w/what-are-the-impacts-of-the-redis-expiration-algorithm (visited on 07/07/2025).

[18] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. "Dynamic Performance Profiling of Cloud Caches". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. Seattle, WA, USA: Association for Computing Machinery, 2014, pp. 1–14. ISBN: 9781450332521. DOI: 10.1145/2670979.2671007. URL: https://doi.org/10.1145/2670979.2671007.

[19] Kia Shakiba and Michael Stumm. "PaperCache: In-Memory Caching with Dynamic Eviction Policies". In: *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*. HotStorage '25. Boston, MA, USA: Association for Computing Machinery, 2025, pp. 107–113. ISBN: 9798400719479. DOI: 10.1145/3736548.3737836. URL: https://doi-org.myaccess.library.utoronto.ca/10.1145/3736548.3737836.

[20] Kia Shakiba, Sari Sultan, and Michael Stumm. "Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation". In: *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. Santa Clara, CA: USENIX Association, Feb. 2024, pp. 89–105. ISBN: 978-1-939133-38-0. URL: https://www.usenix.org/conference/fast24/presentation/shakiba.

[21] Sari Sultan, Kia Shakiba, Albert Lee, Paul Chen, and Michael Stumm. "TTLs Matter: Efficient Cache Sizing with TTL-Aware Miss Ratio Curves and Working Set Sizes". In: *Proceedings of the Nineteenth European Conference on Computer Systems*. EuroSys '24. Athens, Greece: Association for Computing Machinery, 2024, pp. 387–404. ISBN: 9798400704376. DOI: 10.1145/3627703.3650066. URL: https://doi.org/10.1145/3627703.3650066.

[22] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. "RIPQ: Advanced Photo Caching on Flash for Facebook". In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 373–386. ISBN: 978-1-931971-201. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/tang.

[23] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. "Cache Modeling and Optimization using Miniature Simulations". In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, July 2017, pp. 487–498. ISBN: 978-1-931971-38-6. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger.

[24] 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)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 95–110. ISBN: 978-1-931971-201. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger.

[25] Yuchen Wang, Junyao Yang, and Zhenlin Wang. "Dynamically Configuring LRU Replacement Policy in Redis". In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '20. Washington, DC, USA: Association for Computing Machinery, 2021, pp. 272–280. ISBN: 9781450388993. DOI: 10.1145/3422575.3422799. URL: https://doi.org/10.1145/3422575.3422799.

[26] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. "Characterizing storage workloads with counter stacks". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 2014, pp. 335–349.

[27] Ben Wolford. *Everything you need to know about the "Right to be forgotten"*. URL: https://gdpr.eu/right-to-be-forgotten/ (visited on 09/11/2025).

[28] 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)*. USENIX Association, Nov. 2020, pp. 191–208. ISBN: 978-1-939133-19-9. URL: https://www.usenix.org/conference/osdi20/presentation/yang.

[29] 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)*. USENIX Association, Apr. 2021, pp. 503–518. ISBN: 978-1-939133-21-2. URL: https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng.

[30] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. "FIFO queues are all you need for cache eviction". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. Koblenz, Germany: Association for Computing Machinery, 2023, pp. 130–149. ISBN: 9798400702297. DOI: 10.1145/3600006.3613147. URL: https://doi.org/10.1145/3600006.3613147.

[31] Yazhuo Zhang, Juncheng Yang, Yao Yue, Ymir Vigfusson, and K.V. Rashmi. "SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches". In: *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1229–1246. ISBN: 978-1-939133-39-7. URL: https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo.