Segcache: a memory-efficient and scalable in-memory key-value cache for small objects

Juncheng Yang, *Carnegie Mellon University;* Yao Yue, *Twitter;*
Rashmi Vinayak, *Carnegie Mellon University*

Segcache: a memory-efficient and scalable in-memory key-value cache for small objects

Juncheng Yang
Carnegie Mellon University

Yao Yue
Twitter

K. V. Rashmi
Carnegie Mellon University

Abstract

Modern web applications heavily rely on in-memory key-value caches to deliver low-latency, high-throughput services. In-memory caches store small objects of size in the range of 10s to 1000s of bytes, and use TTLs widely for data freshness and implicit delete. Current solutions have relatively large per-object metadata and cannot remove expired objects promptly without incurring a high overhead. We present Segcache, which uses a segment-structured design that stores data in fixed-size segments with three key features: (1) it groups objects with similar creation and expiration time into the segments for efficient expiration and eviction, (2) it approximates some and lifts most per-object metadata into the shared segment header and shared information slot in the hash table for object metadata reduction, and (3) it performs segment-level bulk expiration and eviction with tiny critical sections for high scalability. Evaluation using production traces shows that Segcache uses 22-60% less memory than state-of-the-art designs for a variety of workloads. Segcache simultaneously delivers high throughput, up to 40% better than Memcached on a single thread. It exhibits close-to-linear scalability, providing a close to 8× speedup over Memcached with 24 threads.

1 Introduction

In-memory caches such as Memcached [54] and Redis [13] are widely used in modern web services such as Twitter [18, 46], Facebook [21, 55], Reddit [3] to reduce service latency and improve system scalability. The economy of cache lies within supporting data retrieval more cheaply, and usually more quickly, compared to the alternatives. The usefulness of in-memory caches is judged by their efficiency, throughput, and scalability, given certain hardware resource constraints. Memory efficiency determines the amount of memory a cache needs to achieve a certain miss ratio. Throughput is typically measured in queries per second (QPS) per CPU core. Scalability reflects how well a cache can use multiple cores on a host. There have been several efforts to reduce miss ratio via better eviction algorithms [22, 26, 36, 37]. Many other works focus on improving throughput [41, 51]. However, several other aspects of in-memory caching also play import roles in memory efficiency.

Web services tend to cache small key-value objects in memory, typically in the range of 10s to 1000s of bytes [21, 46].
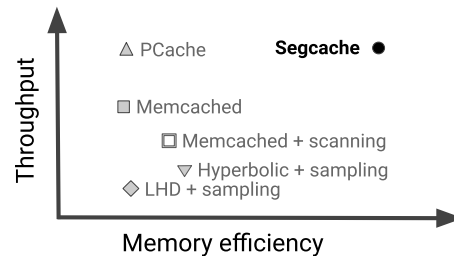

Figure 1: How Segcache compares to state-of-the-art caches

However, most popular production caching systems store a relatively large amount of metadata. For example, both Memcached and Redis impose over 50 bytes of memory overhead per object. Furthermore, research aimed at reducing miss ratio typically ends up expanding object metadata even further [22, 26, 28, 36, 60], as shown in Table 1.

Time-to-live (TTL) is widely used in caching to meet data freshness and feature requirements, or comply with regulations such as GDPR [42, 65–67]. Twitter mandates the use of TTLs in cache, with values ranging from a few minutes to a month. Existing caching systems either remove expired objects lazily or incur high overhead [70] when they attempt to expire more proactively. We summarize the techniques for removing expired objects in Table 2 and discuss them in §2.

Most production in-memory caches use an external memory allocator such as `malloc` or a slab-based memory allocator. The former often subjects the cache to external fragmentation, and the latter to internal fragmentation [63]. In addition, slab-based allocators often suffer from the so-called slab calcification problem [18, 44], or introduce extra cache misses due to slab migration [29, 34].

One way to reduce memory fragmentation is to adopt a log-structured design. This approach has been widely used for file systems [62] and durable key-value stores [14, 31, 33, 57, 58, 63] for their simplicity, high write throughput [51, 62], low fragmentation, and excellent space efficiency [63]. There have been several in-memory caches built with log-structured design. MICA [51], a throughput-oriented system based on one giant log per thread, limits its eviction algorithm to FIFO or CLOCK, both of which are sub-optimal for many

Table 1: Comparison of research systems (all comparisons are with corresponding baselines)

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

workloads. Memshare [37], a multi-tenant caching system, divides DRAM into small logs (called segments) and uses segments to enforce memory partitioning between tenants. However, its computation of miss ratio curve for each tenant and object migration are relatively expensive, which result in reduced throughput compared to Memcached.

As modern servers become denser with CPU cores over time, thread-scalability becomes essential in modern cache design. Several techniques have been proposed to improve scalability in key-value caches and key-value stores, such as static DRAM and data partitioning [50, 51], opportunistic concurrency control with lock-free data structures [32, 41, 52], and epoch-based design [31]. However, each technique comes with its own problems. Static partitioning uses memory inefficiently. Opportunistic concurrency control works better on read-heavy workloads, whereas some caching workloads are write-heavy [46]. An epoch-based system requires a log-structured design with a sub-optimal eviction algorithm.

Achieving high memory efficiency, high throughput, and high scalability simultaneously in caching systems is challenging. Previous works tend to trade one for the other (Fig. 1 and Table 1). In this paper, we present Segcache, a cache design that achieves all the three desired properties. Segcache is a TTL-indexed, dynamically-partitioned, segment-structured[1] cache where objects of similar TTLs are stored in a small fixed-size log called a segment. Segments are first grouped by TTL and then naturally sorted by creation time. This design makes timely removal of expired objects both simple and cheap. As a cache, Segcache performs eviction by merging a few segments into one, retaining only the most important objects, and freeing the rest. Managing the object life cycles at the segment level allows most metadata to be shared within a segment. It also allows metadata bookkeeping to be performed with a limited number of tiny critical sections. These decisions improve memory efficiency and scalability without sacrificing throughput or features.

Below are some highlights of our contributions:

- To the best of our knowledge, Segcache is the first cache design that can efficiently remove all objects immediately after expiration. This is achieved through TTL-indexed, time-sorted segment chains.

---

[1]Since segments are small-sized logs, Segcache can be viewed as a log-structured cache with special properties; see §6 for in-depth comparisons.

Table 2: Techniques for removing expired objects

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

- We propose and demonstrate "object sharing economy", a concept that reduces per-object metadata to just 5 bytes per object, a 91% reduction compared to Memcached, without compromising on features.
- Our single pass, merge-based eviction algorithm uses an approximate and smoothed frequency counter to achieve a balance between retaining high value objects and effectively reclaiming memory.
- We demonstrate that a "macro management strategy", replacing per-request bookkeeping with batched operations on segments, improves throughput. It also delivers close-to-linear CPU scalability.
- Segcache is designed for production on top of Pelikan, and is open sourced (see §4).
- We evaluated Segcache using a wide variety of production traces, and compared results with multiple state-of-the-art designs. Segcache reduces memory footprint by 42-88% compared to Twitter's production system, and 22-58% compared to the best of state-of-the-art designs.

2 Background and Motivation

As a critical component of the real-time serving infrastructure, caches prefer to store data, especially small objects, in DRAM. DRAM is expensive and energy-hungry. However, existing systems do not use the costly DRAM space efficiently. This inefficiency mainly comes from three places. First, existing solutions are not able to quickly remove expired objects. Second, metadata overhead is considerable compared to typical object sizes. Third, internal or external memory fragmentation is common, leading to wasted space. While improvements of admission [24, 25, 39, 40, 43], prefetching [73, 74], and eviction algorithms [22, 23, 26, 30, 36, 37, 49, 68] have been the main focus of existing works on improving memory efficiency [22, 25, 26, 39], little attention has been paid to addressing expiration and metadata reduction [41, 63]. On the

contrary, many systems add more per-object metadata to make smarter decisions about what objects to keep [22, 36, 39, 73].

We summarize recent advancements of in-memory caching systems in Table 1 and discuss them below.

2.1 TTL and expiration in caching

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

2.1.1 The prevalence of TTL

TTLs are used by users of Memcached and Redis [4, 8, 10, 12], Facebook [17], Reddit [3], Netflix [6]. In Twitter's production, *all* in-memory cache workloads use TTLs between one minute and one month. A TTL is specified at write time to determine how long an object should remain accessible in the caching system. An expired object cannot be returned to the client, and a cache miss is served instead.

Cache TTLs serve three purposes. First, clients use TTLs to limit data inconsistency [4, 46]. Writing to the cache is usually best-effort, so it is not uncommon for data in cache and database to fall out of sync. Second, some services use TTLs to prompt periodic re-computation. For example, a recommendation system may only want to reuse cached results within a time window, and recompute periodically to incorporate new activities and content. Third, TTLs are used for implicit deletion. A typical scenario is rate-limiting. Rate limiters are counters associated with some identities. Services often need to cap requests from a particular identify within a predefined time window to prevent denial-of-service attacks. Services store rate limiters in distributed caches with TTLs, so that the counts can be shared among stateless services and reset periodically. Another increasingly common scenario is using TTLs to ensure data in caches comply with privacy laws [42, 67].

2.1.2 Lazy expiration

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

2.1.3 Proactive expiration

Proactive expiration is used to reclaim memory occupied by expired objects more quickly. Although there has been no academic research on this topic to the best of our knowledge, we identified four approaches introduced into production systems over the years, as summarized in Table. 2.

Checking LRU tail is used by Memcached. Before eviction is considered, the system checks a fixed number of objects at the tail of the LRU queue and removes expired objects. Operations on object LRU queues reduce thread scalability due to the extensive use of locking for concurrent accesses [22, 26]. Additionally, this approach is still opportunistic and therefore doesn't guarantee the timely removal of expired objects. Many production caches track billions of objects over a few

LRU queues, so the time for an object to percolate through the LRU queue is very long.

Transient object pool was introduced by Facebook [55]. It makes a special case for the timely removal of objects with small TTLs. The main idea is to store such objects separately, and only allow them to be removed via expiration. However, choosing the TTL threshold is non-trivial and can have side effects [46]. Although Memcached supports it, it is disabled by default.

Full cache scan is a popular approach adopted by Memcached and CacheLib [17]. As the name indicates, this solution periodically scans all the cached objects to remove expired ones. Full cache scan is very effective if the scan is frequent, but it wastes resources on objects that are not expired, which can be the vast majority.

Random sampling is adopted by Redis. The key idea is to periodically sample a subset of objects and remove expired ones. In Redis, if the percentage of the expired objects in the sample is above a threshold, this process continues. While sampling is cheaper per run, the blind nature of sampling decides that it is both inefficient and not very effective. Users have to accept that the sampling can only keep the percentage of expired objects at a pre-configured threshold. Meanwhile, the cost can be higher than full cache scan due to random memory access. There have been some production incidents where Redis could not remove enough expired objects and caused unexpected evictions [70].

Despite the various flaws, proactive expiration is highly regarded by developers of production systems. When asked to replace LRU for a better eviction strategy in Memcached, the maintainer states that "*pulling expired items out actively is better than almost any other algorithmic improvement (on eviction) I could think of.*" [10] Meanwhile, Redis' author mentioned that "*Redis 6 expiration will no longer be based on random sampling but will take keys sorted by expiration time in a radix tree.*" [2]

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

2.2 Object metadata

We observe that the objects stored in in-memory caches are small [46], and the mean object sizes (key+value) of Twitter's top four production clusters are 230, 55, 294, 72 bytes, respectively. This observation aligns with the observations at Facebook [21], and Reddit [3].

Existing systems are not efficient in storing small objects because they store considerable amount of metadata per object. For example, Memcached stores *56 bytes* of metadata with each object [3], which is a significant overhead compared

---

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

[3] 2×8 bytes LRU pointers, 8 bytes hash pointer, 4 bytes access time, 4 bytes expire time, 8 bytes object size, 8 bytes `cas` (compare-and-set, used for atomic update).
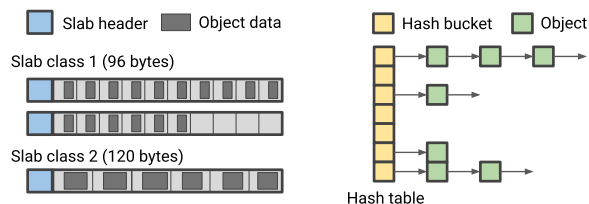
Figure 2: Slab memory allocation (left) and object-chained hash table (right) in Memcached.

to typical object size. All of the metadata fields are critical for Memcached's operations, and cannot be dropped without first removing some functionalities or features.

There have been several attempts at Twitter to cut metadata overhead. For example, Pelikan's slab-based storage removes object LRU queues and reuses one pointer for both hash chain and free object chain. As a result, it reduces object metadata to 38 bytes. However, this prevents Pelikan from applying the LRU algorithm to object eviction, and results in higher miss ratio compared to Memcached in our evaluation. Pelikan also introduced Cuckoo hashing [59] as a storage module for fixed-size objects, only storing 6 bytes (or 14 bytes with `cas`) of metadata per key.

Several academia works have also looked at reducing metadata size. RAMCloud [63] and FASTER [31] use a log-structured design to reduce object metadata. However, their designs target *key-value stores* instead of key-value caches (See discussion in §5). MemC3 [41] redesigns the hash table with Cuckoo hashing and removes LRU chain pointers. However, it does not consider some operations such as `cas` for atomic updates, does not support TTL expiration or other advanced eviction algorithms.

2.3 Memory fragmentation

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

To avoid this problem, other systems such as Memcached use a slab-based memory allocator, allotting a fixed-size slab at a time, which is then explicitly partitioned into smaller chunks for storing objects, as shown in Fig. 2 (left). The chunk size is decided by the `class id` of a slab and configured during startup. A slab-based memory allocator is subjected to internal memory fragmentation at the end of each chunk and at the end of each slab.

Using a slab-based allocator also introduces the slab calcification problem, a phenomenon where some slab classes cannot obtain enough memory and exhibit higher miss ratios. Slab calcification happens because slabs are assigned to classes using the first-come-first-serve method. When popularity among slab classes change over time, the newly popular slab classes cannot secure more memory because all slabs have been assigned. This has been studied in the previous works [29, 44, 46]. Memcached automatically migrates slabs

between classes to solve this problem, however, it is not always effective [2, 5, 9, 15]. Re-balancing slabs may increase the miss ratio because all objects on the outgoing slab are evicted. Moreover, due to workload diversity and complexity in slab migration, it is prone to errors and sometimes causes crash in production [7, 16].

Overall, existing production systems have not yet entirely solved the memory fragmentation problem. Among the research systems, log-structured designs such as MICA [50, 51], memshare [37] and RAMCloud [63] do not have this problem. However, they cannot perform proactive expiration and are limited to using basic eviction algorithms (such as FIFO or CLOCK) with low memory efficiency.

2.4 Throughput and scalability

In addition to memory efficiency, throughput and thread-scalability are also critical for in-memory key-value caches. Memcached's scalability limitation is well documented in various industry benchmarks [11, 55]. The root cause is generally attributed to the extensive locking in the object LRU queues, free object queues, and the hash table. Several systems have been proposed to solve this problem. Some of them remove locking by using simpler eviction algorithms and sacrificing memory efficiency [41, 51]. Some introduces opportunistic concurrency control [41], which does not work well with write-heavy workloads. Some other works use random eviction algorithms to avoid concurrent reads and writes [22, 26], which do not address all the locking contention. Moreover, they reduce throughput due to the large number of random memory accesses.

3 Design principles and overview

The design of Segcache follows three principles.

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

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

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

At a high level, Segcache contains three components: a hash table for object lookup, an object store comprised of segments, and a TTL-indexed bucket array (Fig. 3).

3.1 TTL buckets

Indexing on TTL facilitates efficient removal of expired objects. To achieve this, Segcache first breaks the spectrum of possible TTLs into ranges. We define the time-width of a TTL range t_1 to t_2 ($t_1 < t_2$) as $t_2 - t_1$. All objects in range t_1 to t_2 are treated as having TTL t_1, which is the *approximate TTL* of this range. Rounding down guarantees an object can only be expired early, and no object will be served beyond expiration. Objects are grouped into small fix-sized groups called segments (see next section), and all the objects stored in the same segment have the same approximate TTL. Second, Segcache
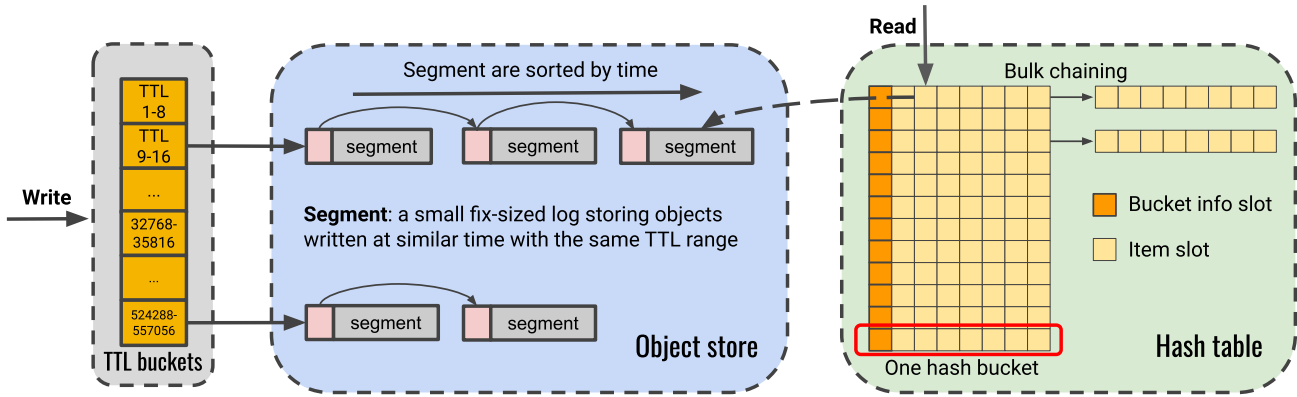
Figure 3: Overview of Segcache. A read request starts from the hash table (right), a write request starts from the TTL buckets (left).

uses an array to index segments based on *approximate TTL*. Each element in this array is called a *TTL bucket*. A segment with a particular approximate TTL value is associated with the corresponding TTL bucket. Within each bucket, segments are chained and sorted by creation time.

To support a wide TTL range from a few second to at least one month without introducing too many buckets or losing resolution on the lower end, Segcache uses 1024 TTL buckets, divided into four groups. From one group to the next, the time-width grows by a factor of 16. In other words, Segcache uses increasingly coarser buckets to efficiently cover a wide range of TTLs without losing relative precision for typical TTL buckets. The boundaries of the TTL buckets are chosen in a way that finding the TTL bucket only requires a few bit-wise operations. We show that this design allows Segcache to efficiently and effectively remove expired objects in §3.5.

3.2   Object store: segments

Segcache uses segments as the basic building blocks for storing objects. All segments are of a configurable size, default to 1 MB. Unlike slabs in Memcached, Segcache group objects stored in the same segment by approximate TTL, not by size. A segment in Segcache is similar to a small log in log-structured systems. Objects are always appended to the end of a segment, and once written, the objects cannot be updated (except for incr/decr atomic operations). However, unlike other log-structured systems [37, 51, 57, 58, 62, 63], where available DRAM is either used as one continuous log or as segments withou no relationship between each other, segments in Segcache are sorted by creation time, linked into chains, and indexed by approximate TTLs.

In Segcache, each non-empty TTL bucket stores pointers to the head and tail of a time-sorted segment chain, with the head segment being the oldest. A write in Segcache first finds the right TTL bucket for the object, and then appends to the segment at the tail of the segment chain. When the tail segment is full, a new segment is allocated. If there is no free segment available, eviction is triggered (§3.6).

3.3   Hash table

As shown in previous works [41], the object-chained hash tables (Fig. 2 (right)) limits the throughput and scalability in the existing production systems [54,71]. Segcache uses a bulk-chaining hash table similar to MICA [51] and Faster [31].

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

Instead of having just one slot per hash bucket, Segcache allocates 64 bytes of memory (one CPU cache line) as eight slots in each hash bucket (Fig. 3). The first slot stores the bucket information, the following six slots store object in-formation. The last slot stores either object information or a pointer to the next hash bucket (when more than seven ob-jects hash to the same bucket). This chaining of hash buckets is called bulk chaining. Bulk chaining removes the need to store hash pointers in the object metadata and improves the throughput of hash lookup by minimizing random accesses.

The bucket information slot stores an 8-bit spin lock, an 8-bit slot usage counter, a 16-bit last-access timestamp , and a 32-bit cas value. Each item slot stores a 24-bit segment id, a 20-bit offset in the segment, an 8-bit frequency counter (described in §3.6.3), and a 12-bit tag. The tag of a key is a hash used to reduce the number of string comparisons when hash collisions happen.

3.4   Object metadata

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

Because objects in the same segment have the same ap-proximate TTL and are written around the same time, Seg-cache computes the approximate expiration time of the whole

segment based on the oldest object in the segment and approximate TTL of the TTL bucket. This approximation skews the clock and incurs early expiration for objects later in the segment. As we will show in our evaluation, early expiration has negligible impact on miss ratio.

Segcache also omits object-level hash chain pointers and LRU chain pointers. Bulk chaining renders hash chain pointer unnecessary. The LRU chain pointers are not needed because because both expiration and eviction are performed at the segment level. Segcache further moves up metadata needed for concurrent accesses (reference counter) into the segment header. In addition, to support `cas`, Segcache maintains a 32-bit `cas` value per hash bucket and shares it between all objects in the hash bucket. While sharing this value may increase false data race between different objects hashed to the same hash bucket, in practice, the impact of this compromise is negligible due to two reasons. First, `cas` traffic is usually orders of magnitude lower than simple read or write, as observed in production environment [46]. Second, one `cas` value is shared only by a few keys, the chance of concurrent updates on different keys in the same hash bucket is small. In the case of a false data race, the client usually retries the request.

The final composition of object metadata in Segcache contains one 8-bit key size, one 24-bit value size, and one 8-bit flag. And Segcache stores only 5 bytes[4] of metadata with each object, which is a 91% reduction compared to Memcached.

3.5 Proactive expiration

In Segcache, all objects in one segment are written sequentially and have the same approximate TTL, which makes it feasible to remove expired objects in bulk. Proactive removal of expired objects starts with scanning the TTL buckets. Because segments linked in each TTL bucket are ordered by creation time and share the same approximate TTL, they are also ordered by expiration time. Segcache uses a background thread to scan the first segment's header in each non-empty TTL buckets. If the first segment is expired, the background thread removes all the objects in the segment, then continues down the chain until it runs into one segment that is not yet expired, at which point it will move onto the next TTL bucket.

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

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

---

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

3.6 Segment eviction

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

Eviction decisions can affect the effectiveness of cache in terms of the miss ratio, thus have been the main focus of many previous works [22, 26, 35, 49, 56, 69]. Segcache does not update objects in-place. Instead, it appends new objects and marks the old ones as deleted. Therefore, better eviction becomes even more critical.

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

3.6.1 Segment selection

The segments merged during each eviction are always from a single TTL bucket. Within this bucket, Segcache merges the first *N consecutive*, *un-expired*, and *un-merged* (in current iteration) segments (Fig. 4). The new segment created from the eviction inherits the creation time of the oldest evicted segment. This design has the following benefits. First, the created segment can be inserted in the same position as the evicted segments in the segment chain, and maintains the time-sorted segment chain property. Second, objects in the created segment still have relatively close creation/expiration time, and the merge distorts their expiration schedules minimally.

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

3.6.2 One-pass merge and segment homogeneity

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

The rationale for retaining a similar number of bytes from each segment is that objects and segments created at a similar time are homogeneous with similar properties. Therefore, no segment is more important than others. Fig. 5a shows the relative standard deviation (RSD, $\frac{std}{mean}$) of the mean object size in consecutive segments and across random segments, and Fig. 5b compares the RSD of live bytes in consecutive and random segments. Both figures demonstrate that consecutive segments are more homogeneous (similar) than random segments. As a result, retaining a similar number of bytes from
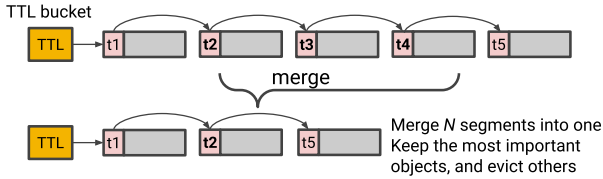
Figure 4: Merge-based segment eviction.

(a) Object size   **(b)** Live bytes

Figure 5: a) Relative standard deviation of mean object size in consecutive segments and random segments. b) Relative standard deviation of live bytes in consecutive segments and random segments.

each is reasonable. However, we remark that the current segment selection and merge heuristics may not be the optimal solution in some cases, and deserve more exploration.

3.6.3 Selecting objects

So far, one question remains unsolved: what objects should be retained in an eviction? An eviction algorithm's effectiveness is determined by its ability to predict future access based on past information. Under the independent reference model (IRM), a popular model used for cache workloads, an object with a higher frequency is more likely to be re-accessed. Moreover, it has been shown in theory that under IRM and for fix-sized objects, the least frequently used (LFU) is k-competitive and the best policy [27, 39, 61, 64].
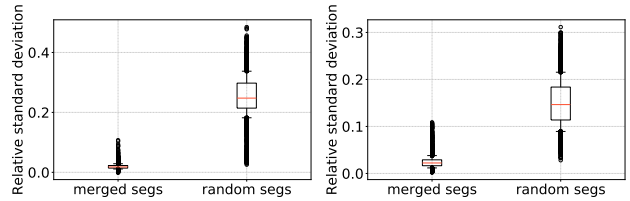
Similar to greedy dual size frequency [35], Segcache uses the frequency-over-size ratio to rank objects. Therefore it needs a frequency counter that is memory-efficient, computationally-cheap, and scalable. Meanwhile, it should allow Segcache to be burst-resistant and scan-resistant. Moreover, The counter needs to provide *higher accuracy for less popular* objects (opposite of the counter-min sketch). This is critical for cache eviction because the highly-popular objects are always retained (cached), and the less popular objects decide the miss ratio of a cache. Segcache uses a novel one-byte counter (stored in hash table), which we call *approximate and smoothed frequency counter* (ASFC), to track frequencies.
Approximate counter. ASFC has two stages. When frequency is smaller than 16 (last four bits of the counter), it always increases by one for every request. In the second stage, it counts frequency similar to a Morris counter [1], which increases with a probability that is the inverse of current value.
Smoothed counter. Segcache uses the last access timestamp, which is shared by objects in the same hash bucket, to rate-limit updates to the frequency counters. The frequency counter for each object is incremented at most once per second. This technique is effective in absorbing sudden request bursts.

Simple LFU is susceptible to cache pollution due to request bursts and non-constant data access patterns. While several approaches such as dynamic aging [20, 39, 61], and window-based frequency [38, 47] have been proposed to address this issue, they require additional parameters and/or extensive tuning [19]. To avoid extra parameters, Segcache resets the frequency of retained objects during evictions, which has a similar effect as window-based frequency.

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

3.7 Thread model and scalability

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

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

4 Implementation and Evaluation

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

- Is Segcache more memory efficient than alternatives?

Table 3: Traces used in evaluation

Trace	Workload type	# requests	TTLs (TTL: percentage)	Write ratio	Mean object size	Production miss ratio
c	content	4.2 billion	1d: 65%, 14d: 27%, 12h: 7%	7%	230 bytes	1-5%
*u*1	user	6.5 billion	5d:1.00	1%	290 bytes	<1%
*u*2	user	4.5 billion	12h:1.00	3%	55 bytes	<1%
n	negative cache	1.6 billion	30d:1.00	2%	45 bytes	∼1%
mix	content + user + negative cache + transient item	11.88 billion	30d: 14%, 14d:11%, 24h: 23%, 12h: 38%, 2min:12%	7%	243 bytes	NA

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

4.1 Implementation

Segcache is implemented as a storage module in the open-sourced Pelikan project. Pelikan is a cache framework developed at Twitter. The Segcache module can both work as a library or be setup as a Memcached-like server. Our current implementation supports multiple worker threads, with a dedicated background thread performing proactive expiration. For our evaluation, eviction is performed by worker threads as-needed, but it is easy to use the same background thread to facilitate background eviction. We provide configurable options to change the number of segments to merge for eviction and segment size. The source code can be accessed at http://www.github.com/twitter/pelikan and archived at http://www.github.com/thesys-lab/segcache.

4.2 Experiment setup

4.2.1 Traces

Single tenant traces. We used week-long unsampled traces from production cache clusters at Twitter (Table. 3, the same as in previous work [46])[5]. Trace *c* comes from a cache storing tweets and their metadata, which is the largest cache cluster at Twitter. Trace *u*1 and *u*2 are both user related, but the access patterns of the two workloads are different, so different TTLs are used. Notably, they are separated into two caches in production because effective and efficient proactive expiration was not achievable prior to Segcache. Trace *n* is a negative result cache, which stores the keys that do not exist in the database, a common way of using cache to shield databases from unnecessary high loads.

Multi tenant trace. Although Twitter's production deployments are single-tenant, multi-tenant deployments are also common because of better resource utilization [21]. To evaluate the performance under multi-tenant workloads, we merged workloads from four types of caches: user, content, negative cache, and transient item cache.

4.2.2 Baselines

Memcached used in our evaluation is version 1.6.6 with segmented LRUs. It supports lazy expiration and checks LRU

tail for expiration. We ran Memcached in two modes, one with cache scanning enabled (s-Memcached), which scans the entire cache periodically to remove expired objects; the other with scanning disabled (Memcached). Other expiration techniques are enabled in both modes. Our evaluation also includes pelikan_twemcache (PCache), Twitter's Memcached equivalent and successor to Twemcache [18]. Compared to Memcached, PCache has a smaller object metadata without LRU queues, and only performs slab eviction [75]. We implemented LHD [22] and Hyperbolic [26] on top of PCache since original implementations are not publicly available. These systems do not consider object expiration. To make the comparisons fairer, we add random sampling to remove expired objects in these two systems, which is also how Redis performs expiration. In the following sections, r-LHD and r-Hyperbolic refer to these enhanced versions. Note that adding random sampling to remove expired objects does not significantly impact the throughput, and we observe less than a 10% difference.

Because we do not modify the networking stack, we focus our evaluation on the storage subsystem. We performed all evaluations by close-loop trace replay on dedicated hosts in Twitter's production fleet using the traces described in §4.2.1. The hosts have dual-socket Intel Xeon Gold 6230R CPU, 384 GB DRAM with one 100 Gbps NIC.

4.2.3 Metrics

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

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

Relative memory footprint. Although miss ratio is a common metric, a sometimes more useful metric is how much memory footprint can be reduced at a certain miss ratio. Therefore, in §4.3, we show this metric using PCache memory footprint as the baseline.

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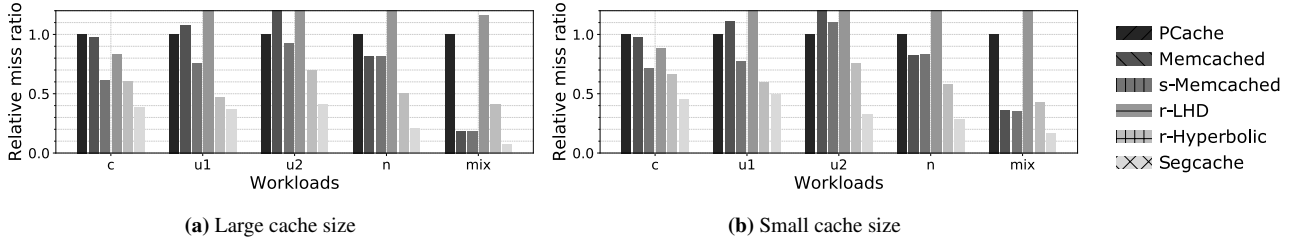
---

[5]The traces are available at http://www.github.com/twitter/cache-trace.

(a) Large cache size   **(b)** Small cache size

Figure 6: Relative miss ratio of different systems (baseline Pelikan is 1), lower is better.

(a) Production miss ratio   **(b)** Miss ratio of Pelikan at the small cache size in Fig.6b
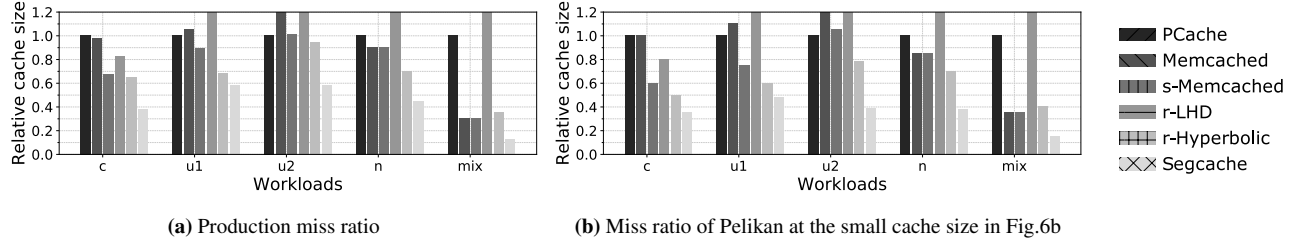
Figure 7: Relative memory footprint to achieve a certain miss ratio, lower is better.

4.3 Memory efficiency

In this section, we compare the memory efficiency of all systems. We present the relative miss ratio at two cache sizes[6] in Fig.6. (1) The "large cache" is the cache size when the miss ratio of Segcache reaches the plateau (<0.05% miss ratio reduction when the cache size increases by 5%). Miss ratios achieved at large cache sizes are similar to production miss ratios. (2) we choose the "small cache" size as 50% of the large cache size.

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

We observe that Memcached and PCache have comparable miss ratios in most workloads (except workload *mix* because PCache is not designed for multi-tenant workloads). While comparing Memcached and s-Memcached, we observe that adding full cache scanning capability significantly reduces the miss ratio by up to 40%, which indicates the importance of proactive expiration. However, as we show in §4.4.1, cache scanning is expensive and reduces throughput by almost half for some workloads. Moreover, we observe that workload *n* and *mix* do not benefit from full cache scanning. Workload *n* shows no benefit because it uses a single TTL of 30 days and no objects expire in the evaluation. Although workload *mix* has a mixture of short and long TTLs, it shows no benefit because the objects of different TTLs are from different workloads with different object sizes, and are stored in different slab classes with different LRU queues. As a result, checking LRU tail for expiration is effective at removing ex-

pired objects and scanning provides little benefit. Overall, we observe that proactively removing expired objects can effectively reduce miss ratio and improve memory efficiency.

State-of-the-art research caching systems, r-LHD and r-Hyperbolic use ranking to select eviction candidates and often reduce miss ratio compared to LRU. In our evaluation, r-Hyperbolic shows lower miss ratio compared to Memcached and PCache, while r-LHD is only better on workload *c*. r-LHD is designed for workloads with a mixture of scan and LRU access patterns (such as block access in storage systems), while in-memory caching workloads rarely show scan requests. This explains why it has higher miss ratios. We have also evaluated r-LHD and r-Hyperbolic without sampling for expiration (not shown), and as expected, they have higher miss ratios due to the wasted cache space from expired objects.

An alternative way of looking at memory efficiency is to determine the cache size required to achieve a certain miss ratio. We show the relative memory footprints of different systems in Fig. 7, using PCache as the baseline. The figures show that for both the production miss ratio and a higher miss ratio, Segcache reduces memory footprint by up to 88% compared to PCache, 60% compared to Memcached, 56% compared to s-Memcached, and 64% compared to r-Hyperbolic.

4.3.1 Ablation study

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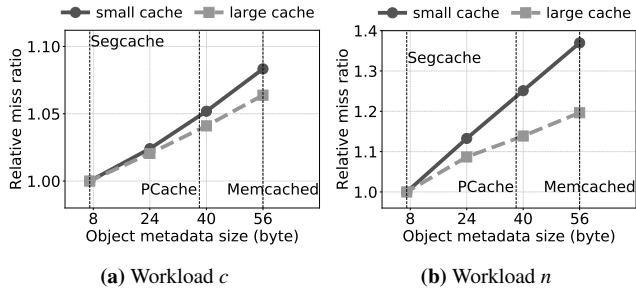
---

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

Figure 8: Impact of object metadata size on miss ratio. Workload *n* has smaller object sizes as compared to workload *c* and hence enjoys larger benefit from reduction from object metadata.

ing the metadata from 56 bytes to 8 bytes reduces the miss ratio by 6-8%. While workload *n* has small object sizes (45 bytes) and reducing object metadata size provides a 20-38% reduction in miss ratio. This result indicates reducing object metadata size is very important, and it is a critical component contributing to Segcache's high memory efficiency.

4.4 Throughput and scalability

4.4.1 Single-thread throughput

Besides memory efficiency, the other important metric of a cache is the throughput. Fig. 9 shows the throughput of different systems. Compared to other systems, PCache and Segcache achieve higher throughput, up to 2.5× faster than s-Memcached, up to 3× faster than r-Hyperbolic, and up to 4× faster than r-LHD. The reason is that PCache performs slab eviction only, and Segcache performs merge-based segment eviction. Both systems perform batched and sequential book-keeping for evictions, which significantly reduces the number of random memory accesses and makes good use of the CPU cache. In addition, PCache and Segcache do not maintain an object LRU chain, which leads to less bookkeeping and also contributes to the high throughput.

Although r-LHD and r-Hyperbolic have lower miss ratios than Memcached, their throughput is also lower. The reason is that both systems use random sampling during evictions, which causes a large number of random memory accesses. One major bottleneck of a high-throughput cache is the poor CPU cache hit ratio, and optimizing CPU cache utilization has been one focus of improving the throughput [51, 52]. Although r-LHD proposes to segregate object metadata for better locality [22], it requires adding more object metadata, and hence would further decrease memory efficiency.

4.4.2 Thread scalability

We show the scalability results in Fig. 10a, where we compare Segcache with Memcached and s-Memcached. Fig. 10a shows that compared to Memcached, Segcache has a higher throughput and close-to-linear scalability. With 24 threads, Segcache achieves over 70 MQPS, a 19.9× boost compared to using a single-thread, while Memcached only achieves 9 MQPS, 3.4× of its single-thread throughput. The reason why Segcache can achieve close-to-linear scalability is the

effect of multiple factors as discussed in §3.7. While there is not much throughput difference between Memcached and s-Memcached, s-Memcached is deadlocked when running with more than 8 threads.

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

4.5 Sensitivity

In this section, we study the effects of parameters in Segcache using workload *c* (from Twitter's largest cache cluster). The most crucial parameter in Segcache is the number of segments to merge for eviction, which balances between processing overhead and memory efficiency. Fig. 10b shows how the miss ratio is affected by the number of merged segments. Compared to retaining no objects (the bar labeled eviction), using merge-based eviction reduces the miss ratio by up to 20%, indicating the effectiveness of merge-based eviction. Moreover, it shows that the point for the minimal miss ratio is between 3 and 4. Merging two segments or more than four segments increases the miss ratio, but not significantly.

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

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

Besides the number of segments to merge, another parameter in Segcache is the segment size. We use the default 1 MB in our evaluation; Fig. 10c shows the impact of different segment sizes. It demonstrates that segment size has little impact on the miss ratio, which is expected. Because the fraction of objects retained from each segment does not depend on the segment size, thus not affecting the miss ratio.
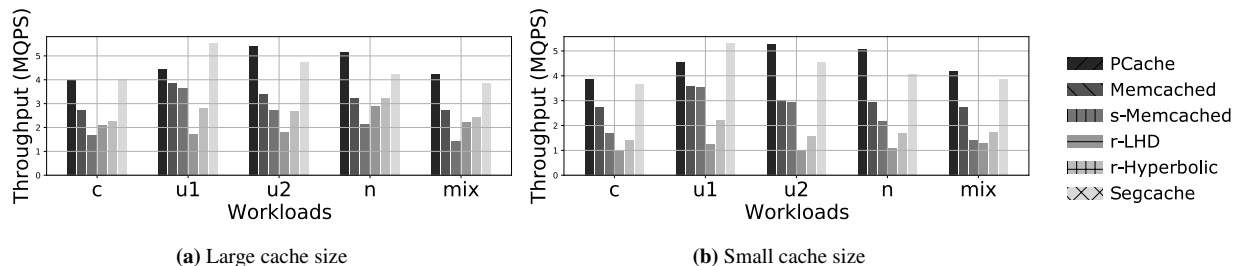
(a) Large cache size   **(b)** Small cache size

Figure 9: Throughput of different systems, the higher the better.

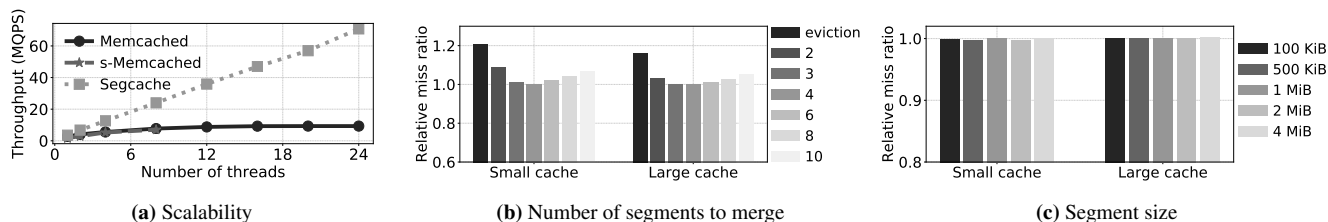**(a)** Scalability   **(b)** Number of segments to merge   **(c)** Segment size

Figure 10: CPU scalability and sensitivity analysis.

5 Discussion

5.1 Alternative proactive expiration designs

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

5.2 In-memory key-value cache vs store

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

TTL. TTLs are far more ubiquitous in caching than in key-value store [3, 6, 12, 17, 55, 70]. We described Twitter's use of TTLs in detail in [46]. In comparison, many datasets are kept in key-value stores indefinitely.

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

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

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

6 Related work

6.1 Memory efficiency and throughput

Approaches for improving memory efficiency fall broadly in the two categories: improving eviction algorithms and adding admission algorithms.

Eviction algorithm. A vast number of eviction algorithms have been proposed in different areas starting from the early 90s [45, 53, 56, 61, 76]. However, most of them focus on the cache replacement of databases or page cache, which are different from a distributed in-memory cache because cached contents in databases and page cache are typically fix-sized blocks with spatial locality. In recent years, several algorithms have been proposed to improve the efficiency of in-memory caching, such as LHD [22], Hyperbolic caching [26], pRedis [60], and mPart [28]. However, all of them add more object metadata and computation, which reduces usable cache size and reduces throughput, which has significant repercussions for caches with small objects.

Segcache uses a merge-based eviction strategy that retains high-frequency small-sized objects from evicted segments, which is similar to a frequency-based eviction algorithms such as LFU [20, 47] and GDSF [35]. However, unlike some of these systems that require parameters tuning, Segcache uses ASFC that avoids these problems. In addition to the eviction algorithm, two major components that contribute to Segcache's low miss ratio is efficient, proactive expiration, and object metadata sharing, which are unique to Segcache.

Admission control. Adding admission control to decide which object should be inserted into the cache is a popular approach for improving efficiency. For example, Adaptsize [25], W-TinyLFU [39], flashshield [40] are designed in the recent years. Admission control is effective for CDN caches, which usually have high one-hit-wonder ratios (up to 30%) with a wide range of object sizes (100s of bytes to 10s of GB). Segcache does not employ an admission algorithm because most of the in-memory cache workloads have low one-hit-wonder ratios (<5%) and relatively small object size ranges. Moreover, adding admission control often add more metadata and extra computation, hurting efficiency and throughput.

Other approaches. There are several other approaches in improving efficiency, such as optimizing slab migration strategy in Memcached [29, 44], compressing cached data [72], and prefetching data [73]. Reducing object metadata size has also been considered in previous works [41]. However, for supporting the same set of functions (including expirations, deletions, `cas`), these approaches need more than twice as much object metadata as Segcache.

Throughput and scalability. A large fraction of works on improving throughput and scalability focus on durable key-value stores [31, 50, 52], which are different from key-value caches as discussed in §5. Segcache is inspired by these works and further improves throughput and scalability by macro management using approximate and shared object metadata.

6.2 Log-structured designs

Segcache's segment-structured design is inspired by several existing works that employ log-structured design [31, 33, 51, 57, 58, 62, 63] in storage and caching systems. The log-structured design has been widely adopted in storage systems to reduce random access and improve throughput. For example, log-structured file system [62] and LSM-tree databases [14, 48] transform random disk writes to sequential writes. Recently log-structured designs have also been adopted in in-memory key-value store [31, 33, 57, 58, 63] to improve both throughput and scalability.

For in-memory caching, MICA [51] uses DRAM as one big log to improve throughput, but it uses FIFO for eviction and does not optimize for TTL expiration. Memshare [37] also uses log-structured design and has the concept of segments. However, Memshare optimizes for multi-tenant cache by moving cache space between tenants to minimize miss ratio based on each tenant's miss ratio curve. Memshare uses a cleaning process to scan N segments, evict one segment, and keep $N-1$ segments where the goal is to enforce memory partitioning between tenants. In terms of performance, scanning N ($N = 100$ in evaluation) segments and *evicting one* incurs a high computation overhead and negatively affects the throughout. Moreover, to compute the miss ratio of different tenants, Memshare adds more metadata to the system, which reduces memory efficiency.

Systems employing a log-structured design benefit from reduced metadata size and memory fragmentation, and increased write throughput, for example, several of the existing works [14, 41, 63] and including Segcache. Compared to these existing works, Segcache achieves a higher memory efficiency by *approximating* and *sharing* object metadata, *proactive TTL expiration*, and using ASFC to retain fewer bytes during eviction while providing a low miss ratio (10% - 25% bytes from each segment are retained in Segcache compared to 75% in RAMCloud [63] and 99% in Memshare [37]).

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

7 Conclusion

Segcache stems out of our insights from production workloads, in particular, the observation that object expiration and metadata play an important role in improving memory efficiency. We chose a TTL-indexed segment-structured design to achieve both high throughput, high scalability and memory efficiency. Our evaluation against state-of-the-art designs from both research and production projects shows that Segcache comes out ahead on our stated goals. Its efficient use of memory bandwidth, near linear scalability, and low-touch configuration poise it favorably as a practical production caching solution suitable for contemporary and future hardware.

References

[1] Approximate counting algorithm. https://en.wikipedia.org/wiki/Approximate_counting_algorithm. Accessed: 2020-08-06.

[2] bit vector + backoff timer + simpler implementation for faster slab reassignment. https://github.com/memcached/memcached/pull/542. Accessed: 2020-08-06.

[3] Caching at reddit. https://redditblog.com/2017/1/17/caching-at-reddit/. Accessed: 2020-05-06.

[4] database caching strategy using redis. https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf. Accessed: 2020-05-06.

[5] Enhance slab reallocation for burst of eviction. https://github.com/memcached/memcached/pull/695. Accessed: 2020-08-06.

[6] Ephemeral volatile caching in the cloud. https://netflixtechblog.com/ephemeral-volatile-caching-in-the-cloud-8eba7b124585. Accessed: 2020-05-06.

[7] Experiencing slab ooms after one week of uptime. https://github.com/memcached/memcached/issue/689. Accessed: 2020-08-06.

[8] Expiration in redis 6. https://news.ycombinator.com/item?id=19664483. Accessed: 2020-08-06.

[9] Faster slab reassignment. https://github.com/memcached/memcached/pull/524. Accessed: 2020-08-06.

[10] Lfu eviction policy. https://github.com/memcached/memcached/issues/543. Accessed: 2020-08-06.

[11] Memcached benchmark. https://github.com/scylladb/seastar/wiki/Memcached-Benchmark. Accessed: 2020-08-06.

[12] Memory used only grows despite unlink/delete of keys. https://github.com/redis/redis/issues/7482. Accessed: 2020-08-06.

[13] Redis. http://redis.io/. Accessed: 2020-05-06.

[14] Rocksdb. https://rocksdb.org/. Accessed: 2020-08-06.

[15] slab auto-mover anti-favours slab 2. https://github.com/memcached/memcached/issue/677. Accessed: 2020-08-06.

[16] slabs: fix crash in page mover. https://github.com/memcached/memcached/pull/608. Accessed: 2020-08-06.

[17] Cachelib: The general-purpose caching engine. 2020.

[18] Chris Aniszczyk. Caching with twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html. Accessed: 2020-08-06.

[19] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.

[20] Martin Arlitt, Rich Friedrich, and Tai Jin. Performance evaluation of web proxy cache replacement policies. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 193–206. Springer, 1998.

[21] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[22] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. Lhd : Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[23] Nathan Beckmann and Daniel Sanchez. Talus: A simple way to remove cliffs in cache performance. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 64–75. IEEE, 2015.

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

[25] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pages 483–498, 2017.

[26] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 499–511, 2017.

[27] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now*

(Cat. No. 99CH36320), volume 1, pages 126–134. IEEE, 1999.

[28] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. mpart: miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, pages 84–95, 2018.

[29] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. Faster slab reassignment in memcached. In *Proceedings of the International Symposium on Memory Systems*, pages 353–362, 2019.

[30] Pei Cao and Sandy Irani. Cost-aware www proxy caching algorithms. In *Usenix symposium on internet technologies and systems*, volume 12, pages 193–206, 1997.

[31] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[32] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. Hotring: A hotspot-aware in-memory key-value store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 239–252, Santa Clara, CA, February 2020. USENIX Association.

[33] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. Flatstore: An efficient log-structured key-value storage engine for persistent memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 1077–1091, New York, NY, USA, 2020. Association for Computing Machinery.

[34] Yue Cheng, Aayush Gupta, and Ali R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery.

[35] Ludmila Cherkasova. *Improving WWW proxies performance with greedy-dual-size-frequency caching policy*. Hewlett-Packard Laboratories, 1998.

[36] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[37] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[38] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *Proceedings of the 19th International Middleware Conference*, pages 94–106, 2018.

[39] Gil Einziger, Roy Friedman, and Ben Manes. Tinylfu: A highly efficient cache admission policy. *ACM Transactions on Storage (ToS)*, 13(4):1–31, 2017.

[40] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 65–78, 2019.

[41] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[42] gdpr. Art. 17 gdpr right to erasure ('right to be forgotten'). https://gdpr-info.eu/art-17-gdpr/. Accessed: 2020-05-06.

[43] Yu Guan, Xinggong Zhang, and Zongming Guo. Caca: Learning-based content-aware cache admission for video content in edge caching. In *Proceedings of the 27th ACM International Conference on Multimedia*, MM '19, page 456–464, New York, NY, USA, 2019. Association for Computing Machinery.

[44] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. {LAMA}: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[45] Song Jiang, Feng Chen, and Xiaodong Zhang. Clockpro: An effective improvement of the clock replacement. In *USENIX Annual Technical Conference, General Track*, pages 323–336, 2005.

[46] Rashmi Vinayak Juncheng Yang, Yao Yue. A large scale analysis of hundreds of in-memory caching clusters at twitter. In *OSDI'20*, 2020.

[47] George Karakostas and D Serpanos. Practical lfu implementation for web caching. *Technical Report TR-622–00*, 2000.

[48] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[49] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[50] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 476–488, 2015.

[51] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. Mica : A holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, 2014.

[52] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. Cache craftiness for fast multicore key-value storage. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 183–196, 2012.

[53] Nimrod Megiddo and Dharmendra S Modha. Arc: A self-tuning, low overhead replacement cache. In *Fast*, volume 3, pages 115–130, 2003.

[54] memcached. memcached - a distributed memory object caching system. http://memcached.org/. Accessed: 2020-05-06.

[55] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.

[56] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[57] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, et al. The case for ramclouds: scalable high-performance storage entirely in dram. *ACM SIGOPS Operating Systems Review*, 43(4):92–105, 2010.

[58] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, et al. The ramcloud storage system. *ACM Transactions on Computer Systems (TOCS)*, 33(3):1–55, 2015.

[59] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[60] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. predis: Penalty and locality aware memory allocation in redis. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 193–205, 2019.

[61] John T Robinson and Murthy V Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS conference on Measurement and modeling of computer systems*, pages 134–142, 1990.

[62] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[63] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured memory for dram-based storage. In *12th {USENIX} Conference on File and Storage Technologies ({FAST} 14)*, pages 1–16, 2014.

[64] Dimitrios N Serpanos and Wayne H Wolf. Caching web objects using zipf's law. In *Multimedia Storage and Archiving Systems III*, volume 3527, pages 320–326. International Society for Optics and Photonics, 1998.

[65] Aashaka Shah, Vinay Banakar, Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Analyzing the impact of GDPR on storage systems. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[66] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. The seven sins of personal-data processing systems under GDPR. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, Renton, WA, July 2019. USENIX Association.

[67] Supreeth Shastri, Melissa Wasserman, and Vijay Chidambaram. Gdpr anti-patterns. *Commun. ACM*, 64(2):59–65, January 2021.

[68] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[69] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. Ripq : Advanced photo caching on flash for facebook. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 373–386, 2015.

[70] Matthew Tejo. Improving key expiration in redis. https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/improving-key-expiration-in-redis.html. Accessed: 2020-08-06.

[71] Twitter. twitter twemcache. https://github.com/twitter/twemcache. Accessed: 2020-05-06.

[72] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–15, 2016.

[73] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 66–79, 2017.

[74] Qiang Yang, Haining Henry Zhang, and Tianyi Li. Mining web logs for prediction models in www caching and prefetching. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 473–478, 2001.

[75] Yao Yue. Eviction strategies. https://github.com/twitter/twemcache/wiki/Eviction-Strategies. Accessed: 2020-08-06.

[76] Yuanyuan Zhou, Zhifeng Chen, and Kai Li. Second-level buffer cache management. *IEEE Transactions on parallel and distributed systems*, 15(6):505–519, 2004.