An Empirical Analysis on Memcached's Replacement Policies

Junyao Yang
junyaoy@mtu.edu
Michigan Technological University
Houghton, Michigan, USA

Yuchen Wang
yuchenwa@mtu.edu
Michigan Technological University
Houghton, Michigan, USA

Zhenlin Wang
zlwang@mtu.edu
Michigan Technological University
Houghton, Michigan, USA

ABSTRACT

The performance of large-scale web services heavily relies on the hit ratio of the key-value caches. One core component of a high-performance key-value cache is the replacement policy. A right replacement policy can help the caching system achieve a better hit ratio with no extra space cost, thereby improving the system's throughput and end-to-end latency. Memcached and Redis are two widely used in-memory key-value caching software in many production systems. Both Memcached and Redis are simple to use and capable of ensuring end-to-end latency requirements for latency critical services. Memcached and Redis use different policies for cache replacement. In contrast, Memcached uses LRU or a variant of segmented-LRU (SegLRU) for replacement, while Redis uses KLRU, a random sampling-based LRU policy which evicts the LRU object from K randomly selected samples. This naturally leads to the question: "how does one compare to the other in actual production usage?" To answer the question, we implement the KLRU policy on Memcached. We evaluate the effectiveness of these three policies using both synthetic and actual production workloads. Our empirical analysis shows that, both SegLRU and KLRU outperform LRU in scalability for write-intensive workloads. However, despite the fact that SegLRU and KLRU are considerably different in terms of their heuristic and implementation, they yield very similar cache hit ratios, throughput, and scalability, with the random sampling-based LRU slightly winning over write-heavy workloads. KLRU also shows advantages in its simplicity in data structures and flexibility in adjusting the sampling size K to adapt to different workloads.

KEYWORDS

Memcached, Replacement Policy, LRU, Random Replacement

1 INTRODUCTION

Modern large-scale web services rely on caching extensively; in-memory key-value (KV) caches are placed between front-end services and back-end storage systems to achieve high throughput and overcome the long latency gap. In-memory KV caches are widely used and discussed in industry and research communities; Memcached [28] and Redis [22] are two in-memory caching solutions commonly deployed in many production environments. Large web service providers like Facebook and Twitter also developed their general-purpose caching frameworks, Cachelib [4] and Pelikan [42], respectively, to handle their caching use cases.

The performance of these caching systems is largely impacted by its replacement/eviction policy, i.e., the algorithm that decides whether an item should be cached or evicted. The Least Recently Used (LRU) is one of the most commonly known replacement policies, which evicts an item based on the item's access recency. Despite its simplicity, the LRU policy has proven quite effective in many caching systems [4, 9, 32, 41]. There are also many other advanced eviction policies, such as [2, 6, 26, 33, 43], which make eviction decisions based on a combination of item's metadata. The effectiveness of an eviction policy primarily depends on two factors:

First, the caching workload; With the rise of cloud and data-driven services, the diversity of in-memory caching workloads has grown drastically compared to the past [1, 4, 10, 41]. Many existing studies have shown that no existing heuristic-based eviction policies can consistently outperform others under every caching use case [26, 33, 43]. As a result, many special purposes caching frameworks have adopted different replacement policies to accommodate different use cases [13, 25, 39, 42].

Second, the underlying cache's structure and the storage medium also constrain the choice of replacement policy. For example, flash-based cache suffers from application-level write amplification (ALWA); even though FIFO does not deliver a good hit ratio, using FIFO helps avoid metadata updates on reads which reduces ALWA [15]. Due to this, FIFO is often the top choice for replacement on a flash-based cache.

Memcached and Redis use replacement policies based on more conservative heuristics to prevent poor performance on unpredictable extreme workload patterns. The early versions of Memcached used the traditional doubly-linked list implementation of LRU as default replacement [12]. The LRU policy helps the cache retain the most recent data; however, the serialized LRU update procedures severely hindered Memcached's multithreaded scalability, especially on write-heavy workloads. To address the thread contention problem, the later version of Memcached (after 1.5.0) added a multi-queue LRU and asynchronous updates, named Segmented LRU (SegLRU), which significantly improved Memcached's performance on write-heavy workloads [12]. Similar multi-list approaches are observed in other caching systems such as [4, 35, 43]. On the other hand, Redis uses random sampling to approximate the true LRU replacement. On cache eviction, the random sampling-based LRU, or KLRU for short, randomly selects K objects from the cache and evicts the least recently used object among the K objects [21]. Such sampling-based technique is commonly used in priority function-based replacement policies [2, 6, 21, 31]. Although both Memcached and Redis use similar replacement heuristics, they are implemented based on very different approaches, which motivates the need for a thorough comparison of their impacts on the in-memory caching system's performance. In this work, we implement Redis-like KLRU in Memcached. Then, we present a detailed comparison based on their impact on Memcached's hit ratios, scalability, throughputs, and latencies. More specifically, this paper is organized as follows:

(1) Section 2 describes the higher-level overview of Memcached and its cache replacement mechanism.

(2) Section 3 presents our design and implementation of KLRU replacements with supports for handling expired items.
(3) Section 4 describes the workloads used in our evaluation.
(4) Section 5.1 compares the miss ratio difference of Memcached configured with SegLRU and KLRU. Our results confirm that both SegLRU and KLRU yield similar miss ratios, especially under a sufficiently large cache.
(5) Section 5.2 presents an empirical performance evaluation for Memcached under SegLRU and KLRU. We observe similar read performance for both policies and slightly higher write performance on KLRU. We also show the impacts of the network latency and global slab allocator lock on Memcached's throughput.

Based on our evaluation, both SegLRU and KLRU perform noticeably better than the early version of Memcached, with KLRU slightly leading SegLRU on write-intensive workloads.

2 MEMCACHED'S REPLACEMENTS OVERVIEW

Memcached is a multithreaded key-value cache that is typically used to reduce latency and increase throughput by caching objects in memory from the slow back-end storage system. Memcached uses a slab-based memory allocator for internal memory management with a default slab size of 1MB. Internally, the memory is divided into slab classes, with each class storing items with the corresponding size. Initially, slabs are distributed to slab classes based on demand. When a slab class exhausts all of its slabs, it will first try to request a new slab; then, if no slab remains, it begins evicting items from the slab class according to the replacement algorithm. The replacement policy and related data structures are thus on a per-class basis.

2.1 LazyLRU Policy

The early version of Memcached(1.4.x) uses the standard doubly-linked list to implement its LRU replacements, where every slab class maintains its LRU list. Memcached uses multiple worker threads to process client requests concurrently. Client requests are commonly in the form of GET or SET requests. When handling a GET request, a worker thread first performs a hash table lookup. If the requested item is found, the worker thread will update the item's metadata and position in the LRU list and then send back the retrieved item to the client. If not found, Memcached will notify the client it's a miss. When handling a SET request, the worker thread first checks whether there is enough free space left for storing the item. The new item will be stored in memory and inserted into the LRU list head if space is sufficient. In case of insufficient memory, the worker thread will trigger the cache replacement/eviction algorithm to remove an old item to make space for the new one.

When the client requests an item, Memcached maintains the LRU list by repositions the newly referenced item into the head of the LRU list. When the slab class exceeds its capacity, it starts evicting items from the tail of the LRU list. These manipulation operations on the LRU list must be serialized to prevent LRU list corruption. To handle that, a dedicated LRU mutex lock is held during LRU list update and eviction so that only one worker thread can modify the LRU list at once. This approach worked at the time, but as Memcached scales to more cores, the LRU lock becomes a bottleneck on Memcached's throughput [37]. One optimization Memcached made to reduce the LRU lock contention is introducing an *item_update_interval*, which is set to one minute by default. The update interval restricts items on the LRU list from being moved to the head of the list more than once per minute. The idea is that, on the LRU list, those recently used items are always closer to the head of the LRU list, which is less likely to age out and be evicted. Thus, constantly updating these recently accessed items on the LRU list is unnecessary. Setting the update interval to one minute drastically decreases the number of LRU list updates, hence reducing LRU lock contention. For convenience, we use LazyLRU[1] to denote the above approach in all of the following discussions.

The LazyLRU approach addresses the excessive LRU locking issue in most read-intensive scenarios (see Section 5.2), but it does not eliminate the problem completely. Workloads with bursts of re-accesses longer than the *item_update_interval* are likely to suffer from performance degradation. Furthermore, the *item_update_interval* only alleviates the LRU locking issue on the read path; for write-intensive workloads, Memcached must hold the LRU lock when inserting a new item into the LRU list.

2.2 SegLRU Policy

The Segmented LRU (SegLRU for short) policy was designed to replace Memcached's original LRU policy (LazyLRU), and it was set as Memcached's default replacement policy starting from 1.5.0 [12]. The Segmented LRU's design is inspired by the OpenBSD's variant of the 2Q algorithm [35].

In the segmented LRU policy, the original LRU list is split into three separate segments: *HOT*, *WARM*, and *COLD*, with each list protected by its own mutex lock. Unlike LazyLRU, where the LRU list updates directly lie on the read request's execution path, the Segmented LRU shifts cached items between/within each segment asynchronously by a background maintainer thread. Thus, it directly avoids potential lock contentions on read-intensive workloads. Next, we briefly outline the semantics of each segment based on the Memcached site post [12]:

(1) HOT behaves like a FIFO queue. A newly arrived item is added to the head of the HOT segment and gradually sinks to the tail of the segment as more items continue to flow into the HOT segment. When the HOT segment reaches its limit, the background maintainer starts shifting the tail item to the head of the Warm segment if it is an active item or the Cold segment if it is inactive. An item is active if it has been re-accessed at least once in the process of the item flowing from segment head to tail.
(2) WARM only admits active items. When the WARM segment reaches its limit, the background maintainer asynchronously moves overflowed inactive items to COLD and re-admits active items back to the head of WARM.
(3) COLD admits inactive items from both HOT and WARM. If an item becomes active in COLD, it will asynchronously move back to the WARM. In case the slab class is full, it starts evicting items from the tail of the COLD.

[1]Note that we use LazyLRU to denote original Memcached(1.4.x)'s default LRU replacement, the LazyLRU is not referring to the work in reference [29]

Here we highlight three main improvements of SegLRU when compared to LazyLRU. (1) Similar to the 2Q replacement [20], SegLRU achieves scan resistance by sinking all inactive items directly to COLD, evicting them from the tail, and using the WARM queue to keep all active items protected. (2) It's more tunable. SegLRU allows you to change the size of the HOT and WARM queues while running. (3) The LRU mutex lock is wholly removed from the read path, thus avoiding all potential waits on the mutex locks for a read request. However, mutexes on the write/update path still exist, which is still a potential bottleneck of scaling Memcached to more cores for write-intensive workloads.

3 KLRU IN MEMCACHED

This section presents our design and implementation of KLRU in Memcached and its impact on background crawling of expired items.

Algorithm 1 Random Sampling Based Eviction in Memcached

```
1:  procedure KLRU_Eviction(cls, K)
2:                              ▷ cls: the slab class
3:                              ▷ K: random sampling size K
4:
5:      LRUItem ← None
6:      LRUTime ← current_time()
7:
8:      while K > 0 do
9:          K ← K − 1
10:         row_index ← rand() mod cls.slabs
11:         col_index ← (rand() mod cls.perslab) ∗ cls.size
12:         item ← cls.slab_list[row_index][col_index]
13:
14:         if item expired then
15:             remove(item)
16:             continue
17:         end if
18:
19:         if (item in HashTable) and (item.time ≤ LRUTime)
    then
20:             LRUItem ← item
21:             LRUTime ← item.time
22:         end if
23:     end while
24:
25:     if (LRUItem ≠ None) then
26:         remove(LRUItem)
27:     end if
28: end procedure
```

3.1 KLRU Design and Implementation

The KLRU replacement is a variant of the LRU algorithm in which, at eviction time, it randomly samples K items and evicts the LRU item among the K items. When $K = 1$, the KLRU is equivalent to random replacement. It's easy to see that as the value of K increases, the probability of selecting a less recently used item also increases. In practice, we notice that when $K = 16$, KLRU behaves nearly identical to the exact LRU replacement. In order to make a fair comparison between Memcached's replacements and KLRU, we implement the KLRU algorithm on top of Memcached, similar to the RankCache design from LHD [2]. On a GET/SET request from the client, KLRU does not maintain the doubly-linked list to keep track of the recency order of items in the cache. Instead, it simply updates the timestamp of the items being referenced. By removing the doubly-linked LRU list, it also no longer needs the LRU mutex lock to safeguard the list manipulation, hence avoiding the potential lock contention problem [37]. Algorithm 1 outlines the steps for the eviction process in KLRU. When the cache is full, worker threads attempt to randomly select K items from the corresponding slab class by generating random indexes for the items (lines 10-12). These selected items are then compared to potential candidates (lines 19-22). Note that only the actual item removal process is serialized (line26) to prevent multiple workers from removing the same item simultaneously. The rest of the algorithm 1 is all done in parallel to maximize the thread concurrency.

Compared to LazyLRU and SegLRU, the most noticeable distinction of the sampling-based replacement policy (KLRU) is it does not rely on any data structure to maintain the ordering of cached items. This unique characteristic of KLRU has its advantages and drawbacks. We summarized three major advantages of KLRU as follows: (1) KLRU uses random sampling for eviction, thus completely eliminating the LRU locks from both read and write paths. (2) The overhead per item is much smaller. KLRU only uses item timestamps during the eviction to compare items' recency. Hence, it saves 16 bytes on the two extra pointers for the doubly-linked LRU list. (3) Random sampling eviction provides great flexibility. Two potentially tunable factors can optimize cache performance under different cache use cases. First, the eviction process described by Algorithm 1 is orthogonal to the item's priority; even though KLRU only uses recency information to decide the lowest priority item on eviction (line 19), the priority function can be easily changed to adapt to even more diverse cache use cases [2, 6, 21]. The second factor is the sampling size K; when workloads favor LRU replacement, one can dynamically increase the value of K (up to 32). When workloads favor random replacement, a smaller value of K can be chosen to increase randomization on the eviction. However, note that this also leads to the drawback of the sampling-based eviction; it does not guarantee that popular or recently accessed items could live longer in the cache, especially for a smaller value of K; thus, it might lead to some occasional latency spike on popular items.

3.2 Handling Item Expiration

Similar to other In-memory caching software [3, 22, 42], Memcached also supports item expiration. The client can specify an item's time-to-live (TTL) on a SET request. The TTL sets the time limit for how long an item will remain valid in the cache before it's expired. Memcached employs a separate background crawler that periodically walks over the LRU list starting from the tail of the list, then removes the expired item and reclaims the memory. However, this approach is not feasible for Memcached with KLRU. The KLRU implementation described earlier completely removes the LRU lists from the system. Thus, the background crawling of expired items can no longer be done by walking through LRU lists. Instead, we

scan for expired items directly at the slab level. Since each slab is a piece of continuous 1MB memory, scanning for expired items directly at the slab level is more cache-friendly than scanning from the LRU list level. After the system runs for a while, items start to be allocated randomly on the slab class slabs. One downside to scanning at the slab level is that it wastes resources for scanning through some unused memory chunks. However, this phenomenon should be rare since in-memory caches are usually full when crawling is triggered. Secondly, Memcached also supports a passive expired item reclaim mechanism triggered whenever eviction occurs. On eviction, Memcached checks a fixed number of items from the tail of LRU list and removes the expired item. Note that this passive mechanism operates on the lock-protected LRU list and lies directly on the eviction execution path; thus, it could potentially limit the scalability due to the heavy locking of the LRU list. To add passive expired item reclaim supports to the KLRU, we adopt a similar passive reclaim mechanism used in Redis [22]. When the KLRU randomly selects K items from the slab class on eviction, we remove all the selected expired items directly (Algorithm 1 line 14-17).

4 TRACES DESCRIPTION

We use three publicly available production workloads and well-known synthetic workloads to evaluate the difference between LazyLRU, SegLRU, and KLRU.

MSR MSR Cambridge traces [34] is a collection of one-week block I/O traces from 13 different enterprise servers, such as web proxy (*prxy*), media server (*mds*), and source control (*src1, src2*), etc. The MSR workload consists of a diverse set of access patterns that previously been used in many caching system evaluations [5, 30, 36, 38, 40].

Twitter Twitter Cache traces [41] are a collection of one-week-long caching traces from 54 Twitter's in-memory caching clusters. The trace suite is around 14 TB in file size and comprises various caching use cases. For our evaluation, we choose sub-traces from 9 different caching clusters, each with approximately 100 million requests and various combinations of read/write ratios.

IBM COS The IBM COS traces [14] consist of 99 traces, and each was collected over a week-long period from IBM's public cloud-based object storage service. Each of these IBM traces has different requests-to-distinct-objects ratios. To ensure that enough items are stored in the cache, we use IBM COS traces containing at least 1 million distinct items and have a steady miss ratio of less than 30%.

Zipf Many prior studies use synthetic traces with Zipfian popularity distribution to approximate real caching workload [1, 2, 7, 10, 11, 17–19, 24]. In this evaluation, we include three synthetic traces that follow Zipfian popularity distribution with Zipfian constants of 0.5 (less skewed), 1.0, and 1.2 (most skewed), respectively, to simulate caching workloads with different levels of popularity skewness.

Our evaluation uses all four workloads to test different cache replacements' miss ratio (or, conversely, the hit ratio) performance. Then, when evaluating the impact of the replacement policy on Memcached's throughput and scalability, we mainly use Twitter's caching workloads, as it contains traces with a variety of read/write ratios.

5 EVALUATION

This section presents miss ratio difference, empirical throughput and latency evaluation for Memcacheds configured with different cache replacement policies, mainly LazyLRU, SegLRU, and KLRU. Our experiments run on a server with two NUMA nodes, each with a 2.20 GHz Intel(R) Xeon(R) CPU E5-2650 v4 containing 12/24 cores/threads with hyper-threading enabled and 250 GB DRAM. The operating system is Fedora 32 with Linux kernel 5.6.15.

In the following evaluation, we use Memcached 1.6.10 with modifications to support KLRU replacement. Moreover, we deploy the Memcached instance only on one NUMA node to eliminate possible memory access latency discrepancies. We use a modified version of mc-crusher [27] on a second node to generate requests for Memcached. We add support for item setbacks on cache misses, as well as support for loading/replaying existing traces on mc-crusher. In Section 5.1, we compared the effectiveness of KLRU and SegLRU in terms of cache miss ratio and concluded that both implementations have their advantage (Figure 3); there is no clear winner from the perspective of miss ratio. Memcached's replacement policies are constructed based on the doubly-linked LRU list, where any modification of the LRU list is protected by mutex lock. In contrast, KLRU is completely lock-free and only requires an update in the timestamp. This section focuses on the impacts of the data structures and locks of the three policies, so we would like to eliminate differences caused by miss ratio. Unless otherwise specified, we over-provision the Memcached memory size so that the entire workload can fit in the cache. Therefore, there will be no capacity misses. Additionally, we always warm the cache for a sufficient time to eliminate discrepancies from the cold start (cold misses) on our results.

5.1 Miss Ratio Comparison

One of the most important metrics to consider when evaluating different cache replacement policies is the cache's miss (or hit) ratio. This section presents the cache's miss ratio results of the workloads mentioned in Section 4. Since Memcached manages eviction independently on each slab class using the same replacement policy, we fix item size for all items in a workload so that all items can fit into exactly one slab class for easier understanding and analysis of the cache's miss ratio.

We generate Miss Ratio Curves (MRC, a plot of cache's miss ratios against cache sizes) for every trace with different replacement policies. When generating MRCs under LazyLRU and SegLRU policies, we use Memcached's default setting for all tunable parameters associated with these policies. For Memcached with KLRU policy, we also do not tune for optimal sampling size K; instead, we conservatively choose K = 16. Figure 1 illustrates the MRCs of 9 representative traces from MSR, Twitter, and IBM COS. When the $item_update_interval$ for LazyLRU is set to high value, LazyLRU behaves similarly to FIFO. Therefore, we have also plotted FIFO MRCs for different traces on figure 1 for reference purposes. Based on the MRCs, we observe that KLRU, LazyLRU, and SegLRU perform very similarly to each other in most instances; they approach steady miss ratios within the roughly same amount of memory size. The largest gap appears in ibm.029, where segLRU results in much higher miss ratios for certain cache sizes. For a more detailed illustration, Figure 2 depicts miss ratio differences between SegLRU

and KLRU for all traces tested under three different cache sizes with the 95%, 75%, and 50% working set sizes, respectively. As expected, with larger cache sizes, miss ratio differences between SegLRU and KLRU are close to zero. We observe more deviation between the two policies when the cache size is small relative to the working set (50%). When the cache size increases, more items, especially those popular items, fit into the cache, resulting in minor differences.

Next, Figure 3 illustrates the impact of cache misses on application performance. This experiment chooses two different traces to show the latency change over time with Memcached configured with SegLRU and KLRU. Miss penalties, i.e., the time to request missed data from the back-end database, are randomly distributed between 100-300us for both tests. To reflect the impact of cache misses on request latencies, we include the cache hit ratio along with the latency plot in Figure 3. The first trace, Figure 3(a), is a synthetic trace that follows Zipfian popularity distribution with alpha = 0.99, interrupted by a long scanning request pattern in the middle. As expected, we observe that SegLRU is much better at protecting the cache from long scanning requests. KLRU appears to recover from the scanning slower than SegLRU, which is reasonable given that we use the sampling size of K=16; at this sampling size, KLRU behaves similarly to an LRU cache. The second trace, Figure 3(b), is IBM COS 029. As shown in Figure 1, the KLRU cache's overall miss ratio is significantly lower than the SegLRU cache's miss ratio under IBM COS 029 trace. We observe a significant latency reduction during the time interval from 2000 to 3000 seconds, where the KLRU cache's hit ratio wins over the SegLRU cache's hit ratio.

Based on our miss ratio comparison results, it's simple to see that there is no clear winner that consistently yields the lowest miss ratio. The replacement policy's effectiveness largely depends on the workload's request pattern. Thus, it's crucial that caching systems, like Memcached, provide runtime tuning capability for their replacement algorithm so that users can manually tune the replacement to better suit their use cases.

5.2 Throughput, Latency, and Scalability

5.2.1 ***Throughput.*** We compare Memcached throughput under three different policies stated in Sections 2 and 3. We configure the Memcached instance with 20 worker threads, each serving 70 connections. We use both read and write-intensive workloads from Twitter in our experiments. Memcached uses per slab class replacement; To make an apple-to-apple comparison on replacement policies, we use the same size (96bytes) for every KV pair so that all items can fit into a single slab class. Figure 4 depicts Memcached throughput under LRU and three other LRU variants. Under the read-intensive case (Fig 4a), Memcached configured with LazyLRU, SegLRU, and KLRU achieves similar throughput. These LRU variants relax LRU locks on the read request, and as a result, they outperform the naive LRU implementation (i.e., locks LRU queue on every read request) by 45%. Under the write-intensive case (Fig 4b), we do not observe any major performance difference between naive LRU and three other variants. Although Memcached configured with KLRU is completely LRU lock-free on write requests, the improvement over throughput compared to naive LRU is insignificant. The similar throughput between Memcached with KLRU and naive LRU

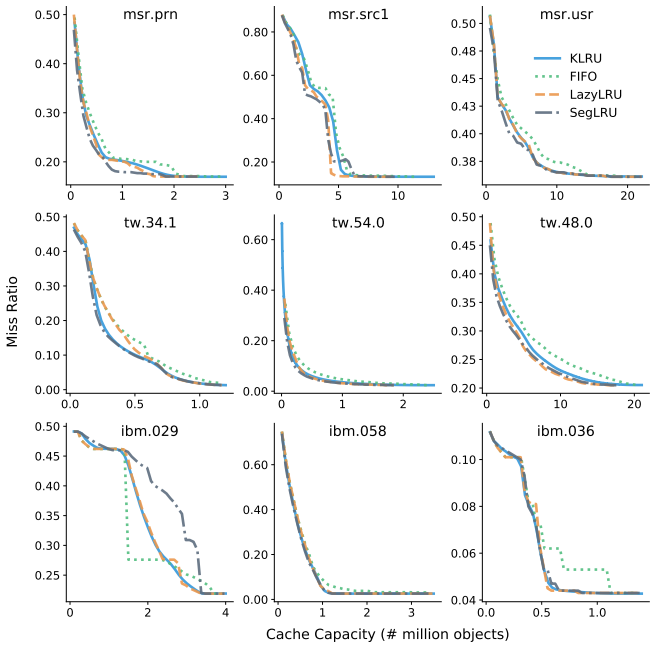

Figure 1: MRCs from MSR, Twitter, and IBM Traces

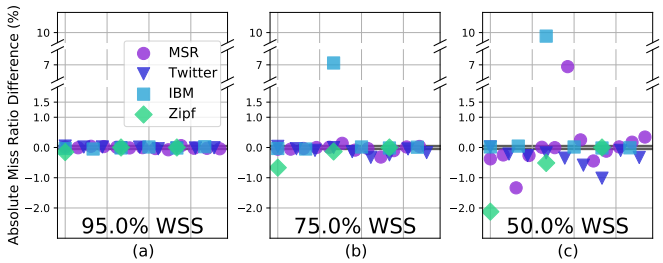

Figure 2: Absolute Miss Ratio Difference (SegLRU Miss Ratio - KLRU Miss Ratio). Positive (Negative) value indicates KLRU (SegLRU) has lower miss ratio.

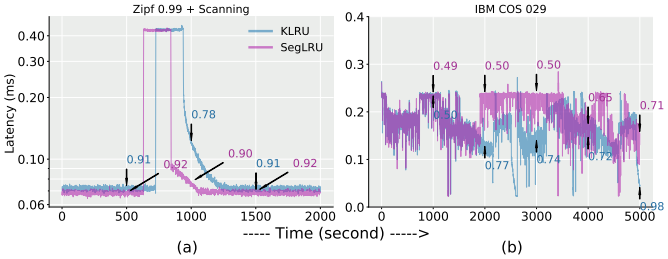

Figure 3: Request Latency v.s. Time. The latency is measured as average latency on every second. The cache's hit ratios, at some time points, are indicated with corresponding colors, respectively. a) shows a workload favors SegLRU. b) shows a workload favors KLRU.

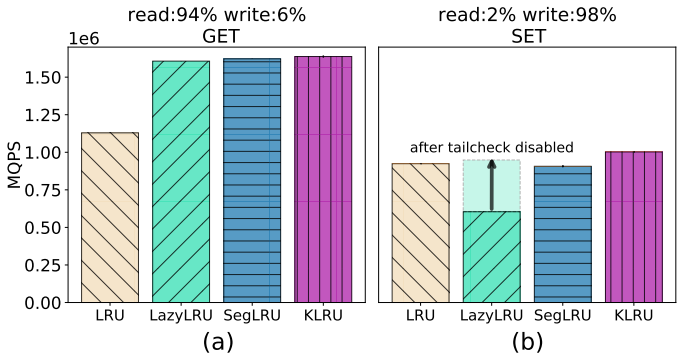

Figure 4: Throughput Comparison on Read-Intensive and Write-Intensive Workload under Different Replacement Policies. The read-intensive trace is Twitter 034 trace, and the write-intensive trace is Twitter 032 trace.

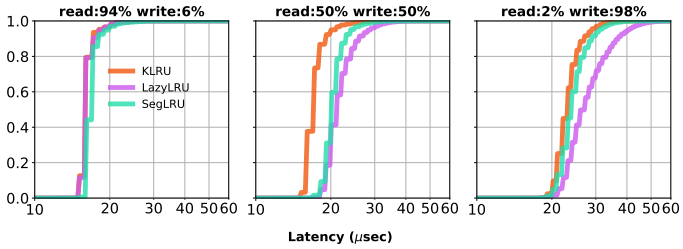

Figure 5: Response Latency Cumulative Distribution The three traces used are Twitter 034, 041, and 032, respectively.

implies that the LRU lock is not the primary limiting factor of Memcached performance under write-intensive workloads. Furthermore, we also observe that Memcached's LazyLRU underperforms naive LRU by 35% in the write-intensive case. Inspecting Memcached's source code, we notice that under LazyLRU, Memcached always attempts to reclaim the memory first before allocating memory for the new item. The memory reclamation is done by walking up a few elements from the LRU tail and removing expired/invalidated items along the walk. This LRU lock-protected reclamation process severely limits LazyLRU's throughput and scalability (Sec. 5.2.3). After removing the reclamation on the write request's execution path, the throughput of LazyLRU recovers to the same level as other variants.

5.2.2 ***Latency.*** Figure 5 illustrates Memcached's latency cumulative distribution under read-intensive, write-intensive, and read/write mixed cases. And Table 1 shows the tail latencies of these three corresponding cases. Memcached shows similar latency distribution and tail behavior for read-intensive workloads under different policies. When the fraction of write requests increases, KLRU and SegLRU show lower end-to-end latency than LazyLRU. In the case of Memcached configured with limited memory, one should expect better tail latency from KLRU as both SegLRU and LazyLRU require mutex lock on eviction, which could hurt tail behavior as Memcached scales.

5.2.3 ***Scalability.*** We present the scalability results in Figure 6, which shows the change in Memcached's throughput as the number of worker threads increases from 1 to 20. Our experiment shows that Memcached scales nearly linearly up to 20 worker threads for all three replacement policies under read-intensive workloads. Memcached's scalability decreases when the workloads shift toward the more write-intensive end. Even though KLRU is totally LRU lock-free, throughputs are still capped at 1 MQPS after 12 worker threads. SegLRU, which locks the LRU queue on write, achieves only slightly lower throughput for the given number of worker threads than KLRU. Similar scalability between KLRU (LRU lock-free write) and SegLRU (LRU locked write) indicates that the LRU lock on write is not the dominating factor that limits Memcached's write capability. For LazyLRU, the throughput degrades when Memcached scales beyond 12 worker threads; As mentioned before, the long LRU-locked reclamation process lies on the write path of LazyLRU creates heavy lock contention as the number of worker threads increases, which bottlenecked Memcached's performance.

In summary, for read-intensive workloads, our experiment shows that all three LRU variants achieve significantly higher throughput than the naive LRU implementation, and all three have close to linear scalability. For write-intensive workloads, Memcached with KLRU shows slightly better performance compared to SegLRU. Nonetheless, we find that Memcached's write capability stops scaling past 12 worker threads regardless of the replacement policies.

5.3 KLRU Analysis

5.3.1 ***Metadata Overheads.*** The KLRU design scrapes off the entire LRU lists layer from Memcached, which leads to two apparent benefits. First, it simplifies the read/write execution path and lowers the overall system complexity. Second, it saves 16 bytes per item by removing two pointers used for the doubly-linked LRU list. For workloads with large item size ($> 1KB$), 16 bytes saving in metadata will not be significant, but for small item size (< 100 bytes) workload, reducing metadata overhead could help save a significant portion of memory resource. For example, under SegLRU, the trace used in Figure 7 would take 96 bytes (including metadata) for each item, but under KLRU, it only takes 80 bytes per item, representing a 17% reduction in total memory consumption.

5.3.2 ***Eviction Overheads.*** Upon eviction, KLRU randomly samples K elements from the slab class and evicts the oldest item among selected K items. Table 2 shows the cost of sampling K items from slab class with K from 1 to 32. The sampling cost grows linearly as the size of K increases. Fortunately, it's been shown that KLRU with a sampling size as small as 16 can very well approximate the actual LRU [2, 40]. In our experiments, we use 16 as the default sampling size. To compare the impacts of eviction overhead with SegLRU, we configure Memcached with three different memory sizes, such that the cache miss ratio was the same for KLRU and SegLRU. Figure 7 shows that the throughput for SegLRU and KLRU is nearly the same in this setting, which indicates that random sampling up to sample size 16 does not negatively impact the Memcached throughput.

5.3.3 ***Reclaiming Expired Items.*** Section 3.2 describes an alternative mechanism for crawling expired items for Memcached

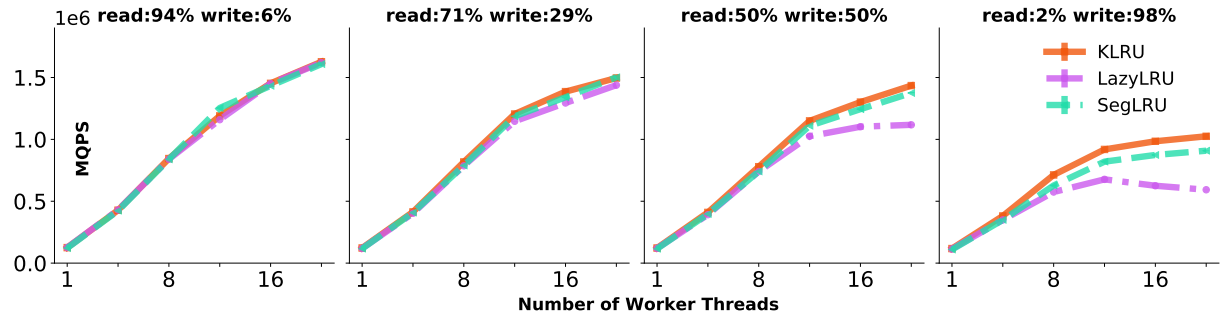

Figure 6: Memcached Thread Scalability under workloads with Different R/W Ratio. The four traces used are Twitter 034, 026, 041, 032, respectively.

Table 1: Memcached Request Tail Latency

	Read Intensive			Mix Read/Write			Write Intensive		
(%)	99	99.9	99.99	99	99.9	99.99	99	99.9	99.99
LazyLRU	21	27	33	34	41	49	48	57	67
SegLRU	23	29	35	29	35	43	36	43	57
KLRU	22	27	33	25	30	37	33	39	48

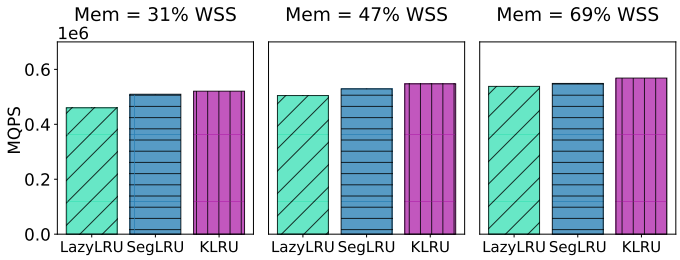

Figure 7: Memcached Throughput under Small Cache Size

Table 2: Eviction Item Sampling Overhead

cost of K random sampling from a slab class						
K	1	2	4	8	16	32
cost (μSec)	.3	.5	1.2	1.9	4.2	6.6

configured with KLRU. In this section, we compare the effectiveness of this *slab-based crawler+passive random sampling reclamation* with the original Memcached implementation. To compare their effectiveness in handling expired items, we use a read-intensive trace with all items' TTL set to 60 seconds so that items that stay in the cache for longer than one minute are considered expired. Figure 8(a) shows Memcached's throughput change over an 8-hour interval. and (b) shows Memcached's memory consumption over the same period. We observe that Memcached with KLRU and SegLRU yield very similar throughputs, with KLRU slightly winning over. In terms of memory consumption, we show that both crawling mechanisms are capable of bounding the memory usage compared to Memcached with expired reclaim disabled.

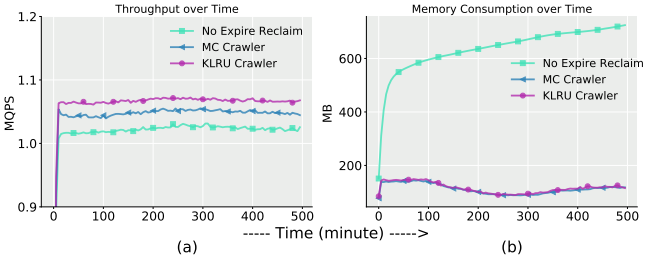

Figure 8: (a) is the Memcached's throughput over time measured every 300 sec. (b) is the Memcached's memory consumption over time measured every 300 sec. The Memcached is configured with 10 worker threads and the trace used here is Twitter 034 trace.

5.4 Impacts of Network Latency and Slab Allocation Lock

In this section, we further compare Memcached replacements by peeling off other performance-limiting factors on Memcached.

We first consider the impact of network latency. Network latency can account for a significant portion of the overall end-to-end request latency, and high network latency can mask the performance differences between different replacement policies. To focus the evaluation on replacement policies, we bypass the network latency by buffering the entire workload into Memcached and then replay requests directly in the process. In Figure 9, we compare Memcached throughputs with and without network bypass enabled. After bypassing the network stack, we observe no major throughput change for Memcached with naive LRU implementation for read-intensive workload, as it is bottlenecked by heavy LRU lock contention, but all three other LRU variants show significant throughput increases.

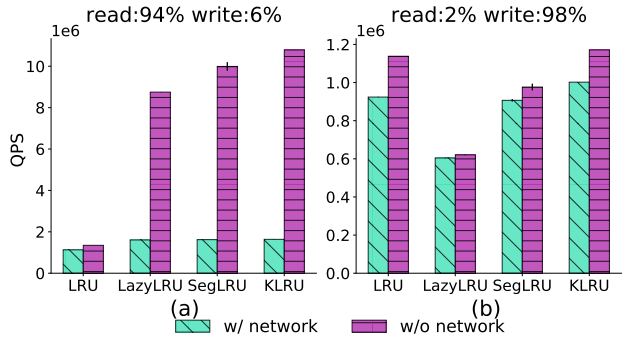

Figure 9: Throughput Differences after Bypass Network

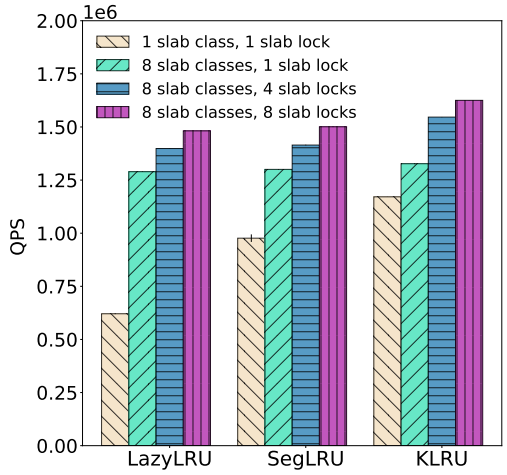

Figure 10: Impacts of Slab Allocator Lock on Memcached with 20 worker threads running Twitter trace 32.0

KLRU's throughput is notably higher than SegLRU and LazyLRU. For the write-intensive workload, Memcached with and without network latency yields similar performance differences among replacement policies, which backs our previous claim (Sec. 5.2.1) that LRU lock is not the major bottleneck on Memcached's write path. Lastly, besides the LRU lock, the Memcached's write path is also guarded by a global slab allocation lock. The slab allocation lock ensures that all internal memory management operations, such as memory allocation and memory reclaim, are serialized. To examine the impacts of slab allocator lock, we temporarily change the global slab lock to the different levels of fine-grained slab locks. Figure 10 shows throughputs of Memcached running write-intensive workload on the first eight default slab classes sharing 1, 4, and 8 slab locks, respectively. As the number of slab locks increases, the contention pressure on each lock decreases, leading to higher throughput.

6 RELATED WORK

There is a great amount of research on cache eviction policies. In this section, we review various policies used in state-of-the-art production caching systems and academia.

Production caching

Facebook - Cachelib [4] is a general-purpose caching engine developed by Facebook in an effort of balancing the generality and specialization of a wide variety of caching systems, including CDN caches, K-V caches, media caches, and social-graph caches. Cachelib is a hybrid cache engine that supports caches composed of DRAM and Flash. Its eviction policy is configurable for the two different underlying storage media. For DRAM cache, each separate DRAM cache pool or slab class is capable of applying different recency or frequency-based eviction policies, including LRU, LRU with multiple insertion points, 2Q [20], and TinyLFU [13]. For Flash cache, FIFO and a pseudo-LRU policy are used in Large Object Cache to amortize the computational cost of flash erasures, while the Small Object Cache only supports policies with no state updates on hits such as FIFO.

Twitter - Segcache [42], a new storage back-end dedicated to small objects developed by Twitter, believes macro management over a contiguous block of object segments could improve both throughput and scalability by reducing CPU cycles on maintaining object indexes for eviction and other operations. It uses a merge-based algorithm to perform eviction by segments. Multiple consecutive, un-expired object segments of the same TTL range are combined into one, and per-object eviction decisions are made while traversing through those segments by evaluating each object's frequency-over-size ratio.

Caffeine [3] is a high performance caching library for Java. It adopts Window TinyLFU eviction policy [13], which involves two cache areas: main cache and window cache. The main cache uses the Segmented LRU eviction policy and TinyLFU admission policy, where the two separate regions of Segmented LRU space are partitioned into 80% of hot items and 20% of non-hot items. The window cache adopts LRU eviction and no admission policy. The size of the main cache and window cache can be adaptively determined by a hill climbing optimization. Their evaluation shows Caffeine's implementation provides a hit rate near Belady's optimal theoretical upper bound over a range of workloads.

Research caching

MemC3 [16] uses Cuckoo hashing, removes Memcached's LRU chain pointers and locks by implementing an approximate LRU cache based on CLOCK replacement algorithm to improve concurrency and throughput, but at the cost of sacrificing memory efficiency, and only works good for workloads with targeted characteristics. MICA [23] adopts an append-only circular log data structure that is write-friendly by placing new objects only at the end of the log. It maps accesses to specific CPU cores and data partitioning to improve scalability and throughput. But it only supports FIFO and approximated LRU policy. HotRing [8], is a Key-Value store with an ordered-ring hash structure that is lock-free to support massive concurrent accesses and improve system throughput.

7 CONCLUSION

This paper presents the results of an empirical study on the performance impacts of two popular LRU implementations on Memcached. Our result reveals that the KLRU implementation, which is LRU (list, lock)-free, results in slightly better write performance and fewer metadata overhead. Our evaluation demonstrates that

both implementations of Memcached exhibit close to linear scalability under read-intensive workloads. However, Memcached's throughput under write-intensive workloads stops scaling after 12 threads, even with the LRU lock-free implementation. This implies that the LRU lock is not currently Memcached's write performance bottleneck. Moreover, we demonstrate that relaxing the global slab allocator lock enhances the write performance, but Memcached's write performance appears to still not scale well with more threads. We believe these findings can offer valuable insights for the development of future in-memory cache designs.

REFERENCES

[1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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* (London, England, UK) *(SIGMETRICS '12)*. Association for Computing Machinery, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[2] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann

[3] Ben-Manes. [n. d.]. caffeine: A high performance caching library for Java. https://github.com/ben-manes/caffeine

[4] 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. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg

[5] Daniel S. Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical Bounds on Optimal Caching with Variable Object Sizes. *Proc. ACM Meas. Anal. Comput. Syst.* 2, 2, Article 32 (jun 2018), 38 pages. https://doi.org/10.1145/3224427

[6] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 499–511. https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein

[7] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 275–290. https://doi.org/10.1145/3183713.3196898

[8] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 239–252. https://www.usenix.org/conference/fast20/presentation/chen-jiqiang

[9] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 379–392.

[10] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. ACM, New York, NY, USA, 143–154.

[11] Diego Didona and Willy Zwaenepoel. 2019. Size-Aware Sharding for Improving Tail Latencies in in-Memory Key-Value Stores. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 79–93.

[12] Dormando. 2018. *Replacing the cache replacement algorithm in memcached.* Retrieved April 10, 2022 from https://memcached.org/blog/modern-lru/

[13] Gil Einziger and Roy Friedman. 2014. TinyLFU: A Highly Efficient Cache Admission Policy. In *2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 146–153. https://doi.org/10.1109/PDP.2014.34

[14] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. 2019. IBM Object Store Traces (SNIA IOTTA Trace Set 36305). In *SNIA IOTTA Trace Repository*, Geoff Kuenning (Ed.). Storage Networking Industry Association. http://iotta.snia.org/traces/key-value?only=36305

[15] Ohad Eytan, Danny Harnik, Effi Ofer, Roy Friedman, and Ronen Kat. 2020. It's Time to Revisit LRU vs. FIFO. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*. USENIX Association. https://www.usenix.org/conference/hotstorage20/presentation/eytan

[16] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan

[17] Bin Fan, Hyeontaek Lim, David G. Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing* (Cascais, Portugal) *(SOCC '11)*. Association for Computing Machinery, New York, NY, USA, Article 23, 12 pages. https://doi.org/10.1145/2038916.2038939

[18] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. 2016. Improving Flash-Based Disk Cache with Lazy Adaptive Replacement. *ACM Trans. Storage* 12, 2, Article 8 (feb 2016), 24 pages. https://doi.org/10.1145/2737832

[19] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 121–136. https://doi.org/10.1145/3132747.3132764

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

[21] Redis Labs. 2020. *Overview of Redis key eviction policies.* Retrieved September 10, 2020 from https://redis.io/docs/reference/eviction/

[22] Redis Labs. 2020. *redis.* Retrieved September 10, 2020 from https://redis.io

[23] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 429–444. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/lim

[24] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies* (Boston, MA, USA) *(FAST'19)*. USENIX Association, USA, 143–157.

[25] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 243–262. https://doi.org/10.1145/3477132.3483568

[26] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache

[27] memcached. 2020. *mc-crusher.* https://github.com/memcached/mc-crusher Accessed: 2022-05-15.

[28] memcached. 2020. *memcached.* Retrieved September 10, 2020 from https://memcached.org

[29] Shane Hansen Alexa Griffith Mike Hurwitz, Shray Kumar. 2022. *LazyLRU: An in-memory cache with limited locking.* https://github.com/TriggerMail/lazylru

[30] Cheng Pan, Xiaolin Wang, Yingwei Luo, and Zhenlin Wang. 2021. Penalty- and Locality-Aware Memory Allocation in Redis Using Enhanced AET. *ACM Trans. Storage* 17, 2, Article 15 (may 2021), 45 pages. https://doi.org/10.1145/3447573

[31] K. Psounis and B. Prabhakar. 2002. Efficient randomized Web-cache replacement schemes using samples from past eviction times. *IEEE/ACM Transactions on Networking* 10, 4 (2002), 441–454. https://doi.org/10.1109/TNET.2002.801414

[32] Guocong Quan, Jian Tan, Atilla Eryilmaz, and Ness Shroff. 2019. A New Flexible Multi-Flow LRU Cache Management Paradigm for Minimizing Misses. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 2, Article 39 (jun 2019), 30 pages. https://doi.org/10.1145/3341617.3326154

[33] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 341–354. https://www.usenix.org/conference/fast21/presentation/rodriguez

[34] SNIA. 2020. *MSR Cambridge Traces.* http://iotta.snia.org/traces/388 Accessed: 2020-03-15.

[35] Ted Unangst. 2014. *2Q Buffer Cache in OpenBSD.* Retrieved April 10, 2022 from https://undeadly.org/cgi?action=article;sid=20140908113732

[36] Yuchen Wang, Junyao Yang, and Zhenlin Wang. 2021. Dynamically Configuring LRU Replacement Policy in Redis. In *The International Symposium on Memory Systems* (Washington, DC, USA) *(MEMSYS 2020)*. Association for Computing Machinery, New York, NY, USA, 272–280. https://doi.org/10.1145/3422575.3422799

[37] Alex Wiggins and Jimmy Langston. 2012. Enhancing the scalability of memcached. *Intel document* (2012). https://www.intel.com/content/dam/develop/

external/us/en/documents/memcached-05172012.pdf

[38] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. 2014. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 335–349. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires

[39] Gang Yan, Jian Li, and Don Towsley. 2021. Learning from Optimal Caching for Content Delivery. In *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies* (Virtual Event, Germany) *(CoNEXT '21)*. Association for Computing Machinery, New York, NY, USA, 344–358. https://doi.org/10.1145/3485983.3494855

[40] Junyao Yang, Yuchen Wang, and Zhenlin Wang. 2021. Efficient Modeling of Random Sampling-Based LRU *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 32, 11 pages. https://doi.org/10.1145/3472456.3472514

[41] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. 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, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[42] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. 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, 503–518. https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng

[43] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 91–104.