A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter

JUNCHENG YANG, Carnegie Mellon University, USA
YAO YUE, Twitter, USA
K. V. RASHMI, Carnegie Mellon University, USA

Modern web services use in-memory caching extensively to increase throughput and reduce latency. There have been several workload analyses of production systems that have fueled research in improving the effectiveness of in-memory caching systems. However, the coverage is still sparse considering the wide spectrum of industrial cache use cases. In this work, we significantly further the understanding of real-world cache workloads by collecting production traces from 153 in-memory cache clusters at Twitter, sifting through over 80 TB of data, and sometimes interpreting the workloads in the context of the business logic behind them. We perform a comprehensive analysis to characterize cache workloads based on traffic pattern, time-to-live (TTL), popularity distribution, and size distribution. A fine-grained view of different workloads uncover the diversity of use cases: many are far more write-heavy or more skewed than previously shown and some display unique temporal patterns. We also observe that TTL is an important and sometimes defining parameter of cache working sets. Our simulations show that ideal replacement strategy in production caches can be surprising, for example, FIFO works the best for a large number of workloads.

CCS Concepts: • **Information systems** → **Information storage systems;**

Additional Key Words and Phrases: Cache, in-memory key-value cache, key-value store, workload analysis, datasets, Twitter

ACM Reference format:
Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. A Large-scale Analysis of Hundreds of In-memory Key-value Cache Clusters at Twitter. *ACM Trans. Storage* 17, 3, Article 17 (August 2021), 35 pages.
https://doi.org/10.1145/3468521

1 INTRODUCTION

In-memory caching systems such as Memcached [15] and Redis [18] are heavily used by modern web applications to reduce accesses to storage and avoid repeated computations. Their popularity has sparked a lot of research, such as reducing miss ratio [30, 35, 47, 48], or increasing throughput and reducing latency [59, 72, 74, 80]. However, the effectiveness and performance of in-memory caching can be workload dependent. And several important workload analyses against production

This work was supported in part by NSF Grants No. CNS 1901410 and No. CNS 1956271, in part by Facebook Distributed Systems Research grant, and in part by Facebook PhD Fellowship.

Authors' addresses: J. Yang and K. V. Rashmi, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA; emails: {juncheny, rvinayak}@cs.cmu.edu; Y. Yue, Twitter, 1355 Market Street, San Francisco, CA 94103, USA; email: yao@twitter.com.

1553-3077/2021/08-ART17 $15.00
https://doi.org/10.1145/3468521

systems [28, 84] have guided the explorations of performance improvements with the right context and tradeoffs in the past decade [59, 80].

Nonetheless, there remains a significant gap in the understanding of current in-memory caching workloads. First, there has been a lack of comprehensive studies covering the wide range of use cases in today's production systems. Second, there have been new trends in in-memory caching usage since the publication of previous work [28]. Third, some aspects of in-memory caching received little attention in the existing studies, but are known as critical to practitioners. For example, TTL is an important aspect of configuring in-memory caching, but it has largely been overlooked in research. Last, unlike other areas where open-source traces [89, 90, 105, 107] or benchmarks [49] are available, there has been a lack of open-source in-memory caching traces. Researchers have to rely on storage caching traces [30], key-value database benchmarks [59, 80], or synthetic workloads [51, 81] to evaluate in-memory caching systems. Such sources either have different characteristics or do not capture all the characteristics of production in-memory caching workloads. For example, key-value database benchmarks and synthetic workloads do not consider how object size distribution changes over time, which impacts both miss ratio and throughput of in-memory caching systems.

In this work, we bridge this gap by collecting and analyzing workload traces from 153 Twemcache [6] clusters at Twitter, one of the most influential social media companies known for its real-time content. Our analysis sheds light on several vital aspects of in-memory caching overlooked in existing studies and identifies areas that need further innovations. The traces used in this article are made available to the research community [1]. To the best of our knowledge, this is the first work that studied over 100 different cache workloads covering a wide range of use cases. We believe these workloads are representative of cache usage at social media companies and beyond, and hopefully provide a foundation for future caching system designs. Here is a summary of our discoveries:

(1) In-memory caching does not always serve read-heavy workloads, write-heavy (defined as write ratio > 30%) workloads are very common, occurring in more than 35% of the 153 cache clusters we studied.
(2) TTL must be considered in in-memory caching, because it limits the effective (unexpired) working set size. Efficiently removing expired objects from cache needs to be prioritized over cache eviction.
(3) In-memory caching workloads follow approximate Zipfian popularity distribution, sometimes with very high skew. The workloads that show the most deviations tend to be write-heavy workloads.
(4) The object size distribution is not static over time. Some workloads show both diurnal patterns and experience sudden, short-lived changes, which pose challenges for slab-based caching systems such as Memcached.
(5) Under reasonable cache sizes, FIFO often shows similar performance as LRU, and LRU often exhibits advantages only when the cache size is severely limited.

These findings provide a detailed new look into production in-memory caching systems, while unearthing some surprising aspects not conforming to the folklore and to the commonly used assumptions.

2 IN-MEMORY CACHING AT TWITTER

2.1 Service Architecture and Caching

Twitter started its migration to a service-oriented architecture, also known as microservices, in 2011 [8]. Around the same time, Twitter started developing its container solution [2, 3] to support

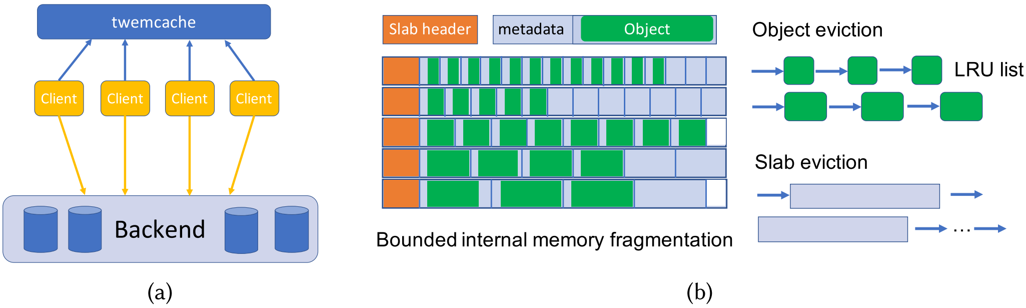


Fig. 1. (a) Twemcache is used as a look-aside cache: upon cache miss, cache clients read from the backends then write to cache; upon cache writes, clients write to backends, then write into cache. (b) Slab-based memory management for bounded memory fragmentation. While Memcached uses object eviction, Twemcache uses slab eviction, which evicts all objects in one slab and returns the slab to global pool.

the impending wave of services. Fast forward to 2020, the real-time serving stack is mostly service-oriented, with hundreds of services running inside containers in production. As a core component of Twitter's infrastructure, in-memory caching has grown alongside this transition. Petabytes of DRAM and hundreds of thousands of cores are provisioned for caching clusters, which are containerized.

At Twitter, in-memory caching is a managed service, and new clusters are provisioned semi-automatically upon request (Figure 1(a)). There are two in-memory caching solutions deployed in production, Twemcache, a fork of Memcached [15], is a key-value cache providing high throughput and low latency. The other solution, named Nighthawk, is Redis-based and supports rich data structures (list, set and map) and replication for high availability. In this work, we focus on Twemcache, because it serves the majority of cache traffic.

Cache clusters at Twitter are considered *single-tenant*[1] based on the service team requesting them. This setup is very beneficial to workload analysis, because it allows us to tag use cases, collect traces, and study the properties of workloads individually. A multi-tenant setup will make similar study extremely difficult, as researchers have to tease out individual workloads from the mixture, and somehow connect them to their use cases. In addition, smaller but distinct workloads can easily be overlooked or mis-characterized due to low traffic.

Unlike other cache cluster deployments, such as social graph caching [32, 37] or CDN caching [65, 106], Twemcache is mostly deployed as a single-layer cache, which allows us to analyze the requests directly from clients without being filtered by other caches. Previous work [65] has shown that layering has an impact on properties of caching workloads, such as popularity distribution. This single-tenant, single-layer design provides us the perfect opportunity to study the properties of the workloads.

Twemcache is used as a look-aside cache similar to the Memcached deployments at Facebook [84]. For a look-aside cache, cache clients directly fetch from the backend during cache misses rather than the cache fetching from the backend. On the read path with a cache hit, the object is served from Twemcache. Upon a cache miss, the client retrieves the object from the backend and then writes it into Twemcache. On the write path, the client first writes data into the backend (if there is a backend for the cache), and then writes into the cache. Look-aside cache design allows cache to be oblivious to the backend designs, which simplifies the cache design. This is important,

[1] Although each cluster is single-tenant, each tenant might cache multiple types of objects of different characteristics.

because the backend of an in-memory key-value cache can be a database, or a computation service or at times no backend is used as described later.

2.2 Twemcache Provisioning

There are close to 200 Twemcache clusters in each data center as of writing. Twemcache containers are highly homogeneous and typically small, and a single host can run many of them. The number of instances provisioned for each cache cluster is computed from user inputs including throughput, estimated dataset sizes, and fault tolerance.

It is easy to see that throughput dictates the number of cores, and dataset size dictates the required cache size. However, cache provisioning considering fault tolerance is less intuitive, because Twemcache does not support replication for efficiency and latency reasons. In this case, the fault tolerance level of a Twemcache cluster indicates the max percentage of computation/cache capacity can fail at any instant. For example, in production, all Twemcache clusters have this parameter smaller than or equal to 5%, meaning at most 5% of instances can fail at any moment. This dictates the number of instances of each cluster to be at least 20 to be fault-tolerant (since otherwise, losing one instance will lead to loss of more than 5% of the planned capacity).

The final number of instances of each cluster is automatically calculated first by identifying the correct bottleneck (core/DRAM/fault tolerance) and then applying other constraints, such as number of connections to support. Size of production cache clusters ranges from 20 to thousands of instances.

2.3 Overview of Twemcache

Twemcache forked an earlier version of Memcached with some customized features. In this section, we briefly describe some of the key aspects of its designs.

Slab-based memory management. Twemcache often stores small and variable-sized objects in the range of a few bytes to 10 s of KB. On-demand heap memory allocators such as ptmalloc [61], jemalloc [13] can cause large and unbounded external memory fragmentation [92], which is highly undesirable in production environment, especially when using smaller containers. To avoid this, Twemcache inherits the slab-based memory management from Memcached (Figure 1(b)). Memory is allocated as fixed size chunks called *slabs*, which default to 1 MB. Each slab is then evenly divided into smaller chunks called *items*. The *class* of each slab decides the size of its items. By default, Twemcache grows item size from a configurable minimum (default to 88 bytes) to just under a whole slab. The growth is typically exponential, controlled by a floating point number called growth factor (default to 1.25), though Twemcache also allows precise configuration of specific item sizes. Higher slab classes correspond to larger items. An object is mapped to the slab class that best fits it, including metadata. In Twemcache, this per-object metadata is 49 bytes. For example, a slab of class 12 has 891 items of 1,176 bytes each, and each item stores up to 1,127 bytes of key plus value. Slab-based allocator eliminates external memory fragmentation at the cost of bounded internal memory fragmentation.

Eviction in slab-based cache. To store a new object, Twemcache first computes the slab class by object size. If there is a slab with at least one free item in this slab class, then Twemcache uses the free item. Otherwise, Twemcache tries to allocate a new slab into this class. When cache is full, slab eviction is needed for allocation.

Some caching systems such as Memcached primarily performs item-level eviction, which happens in the same slab class as the new object. Memcached uses an approximate LRU queue per slab class to track and evict the least recently used item. This works well as long as object size distribution remains static. However, this is often not true in reality, as we show in Section 4.7.2. For example, if all keys start with small values that grow over time, then new writes will eventually require objects to be stored in a higher slab class. Another common example is that objects accessed and created during day and night can have different sizes showing a diurnal pattern. If all memory has been allocated when this happens, then there will be effectively no memory to give out.

This problem is called *slab calcification* and is further explored in Section 4.7.2. Memcached developed a series of heuristics to move memory between slab classes, and yet they have been shown as non-optimal [10, 11, 20, 64] and error prone [9].

To avoid slab calcification, Twemcache uses slab eviction (Figure 1(b)). This allows the evicted slab to transition into any other slab class. There are three approaches to choose the slab to evict: choosing a slab randomly (random slab), choosing the least recently used slab (slabLRU), and choosing the least recently created slab (slabLRC). In addition to avoiding slab calcification, slab-only eviction removes two pointers from object metadata compared to Memcached. We further compare object eviction and slab eviction in Section 7.

Hot object tracking and client caching. Load imbalance is a common problem in distributed systems, especially for stateful services such as cache [23, 54, 60, 66, 81, 87]. Based on the observation that most services can tolerate some level of short inconsistency, Twemcache tracks hot objects using a sliding window with sampling. An object is labeled as a hot object if both the object's frequency in the window and the cache's request rate are above threshold. Once Twemcache identifies a hot object, it signals the hot object to the client, and the client will cache the response locally for some time and avoid sending more requests for this object. Using this approach, Twemcache effectively reduces load imbalance and load spikes caused by hot objects.

2.4 Cache Use Cases

At Twitter, it is generally recognized that there are three main use cases of Twemcache: caching for storage, caching for computation, and caching for transient data. We remark that there is no strict boundary between the three categories, and production clusters are not explicitly labeled. Thus the percentages given below are rough estimates based on our understanding of each cache cluster and their corresponding application.

2.4.1 Caching for Storage. Using cache to facilitate reading from storage is the most common use case. Backend storage such as databases usually has a longer latency and a lower bandwidth than in-memory cache. Therefore, caching these objects reduce access latency, increases throughput, and shelters the backend from excessive read traffic. This use case has received the most attention in research. Several efforts have been devoted to reducing miss ratio [30, 31, 35, 47, 48, 53, 65, 109], redesigning for a denser storage device to fit larger working sets [32, 42, 55, 69, 75, 76, 95, 98], improving load balancing [41, 44, 51, 60, 63, 81, 114], and increasing throughput [40, 59, 79, 80].

As shown in Figure 2, although only 30% of the clusters fall into this category, they account for 65% of the requests served by Twemcache, 60% of the total DRAM used, and 50% of all CPU cores provisioned.

2.4.2 Caching for Computation. Caching for computation is not new—using DRAM to cache query results has been studied and used since more than two decades ago [24, 82]. As real-time stream processing and **machine learning (ML)** become increasingly popular, an increasing number of cache clusters are devoted to caching computation-related data, such as features, intermediate and final results of ML prediction, and so-called object hydration—populating objects with additional data, which often combines storage access and computation.

Overall, caching for computation accounts for 50% of all Twemcache clusters in cluster count, 26%, 31%, and 40% of request rate, cache sizes, and CPU cores.

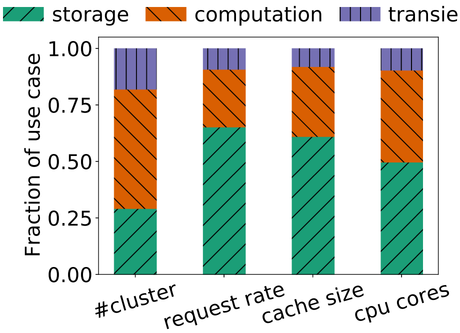


Fig. 2. Resources consumed for the three cache use cases.

2.4.3 Transient Data with No Backing Store. The third typical cache usage evolves around objects that only live in cache, often for short periods of time. It is not caching in the strict sense and therefore has received little attention. Nonetheless, in-memory caching is often the only production solution that meets both the performance and scalability requirements of such use cases. While data loss is still undesirable, these use cases really prize speed, and tolerate occasional data loss well enough to work without a fallback.

Some notable examples are rate limiters, deduplication caches, and negative result caches. Rate limiters are counters associated with user activities. They track and cap user requests in a given time window and prevent denial-of-service attacks. Deduplication caches are a special case of rate limiters, where the cap is 1. Negative result caches store keys from a larger database that are known to be misses against a smaller, sparsely populated database. These caches short-circuit most queries with negative results, and drastically reduce the traffic targeting the smaller database.

In our measurements, 20% of Twemcache clusters are under this category. Their request rates and cache sizes account for 9% and 8% of all Twemcache request rates and cache sizes, meanwhile, they account for 10% of all CPU cores of Twemcache clusters.

3 METHODOLOGY

3.1 Log Collection

Lockless command logging (klog). To avoid unpredictable high tail latency caused by lock contention and blocking I/O in persisting logs, Twemcache uses lockless command logging. The log format is compatible with W3C Common Logfile Format [14], including fields for *timestamp*, *client*, *request*, *return status*, and *response size*. Further, the request contains *command* and *object id* for all requests, and *flags* and *TTL* for write requests. The formatted requests are appended to a thread-local circular buffer, which are then collected by a background thread.

To control how quickly log files grow and get recycled, Twemcache logs one request every R requests, where R is a parameter adjustable at runtime. This log sampling technique is often referred to as temporal sampling. It is useful for investigating workload behavior such as hotkeys, because it can cover the full keyspace, and the frequency of hot objects can be accurately recovered from the log. That said, our trace collection contains full traces without temporal sampling. R was set to 1 on the instances we chose to log *every* request during the collection period. Collecting unsampled traces allows us to avoid drawing potentially biased conclusions caused by sampling. Moreover, we chose to collect traces from two instances of every cache cluster both to prevent possible cache failure during log collection and to compare results between instances for higher fidelity. Barring cache failures, the two instances have no overlapping keys.

3.2 Privacy Preservation

Twemcache clients are frontend servers instead of end-users, so the logs cannot be mapped to Twitter users. Besides, we do not log value (response), and the request keys are anonymized. Moreover, throughout our analysis, we do not focus on any specific keys, instead, we only look at the statistics of request stream and the cache. In addition, we kept all raw logs inside Twitter data warehouse during the analysis, and have deleted them at the time of writing following GDPR guidelines.

3.3 Log Overview

We collected around 700 billion requests (80 TB in raw file size) from 306 instances of 153 Twemcache clusters, which include all clusters with per-instance request rate more than 1,000 **queries-per-sec (QPS)** at the time of collection. To simplify our analysis and presentation, we focused on the 54 largest caches, which account for 90% of aggregated QPS and 76% of allocated memory. In the following sections, we use Twemcache workloads to refer to the workloads from these 54 Twemcache clusters. Although we only present the results of these 54 caches, we performed the same analysis on the smaller caches, and they do not change our conclusions.

4 PRODUCTION STATS AND WORKLOAD ANALYSIS

In this section, we start by describing some common production metrics to provide a foundation for our discussion, and then move on to workload analyses that can only be performed with detailed traces.

4.1 Miss Ratio

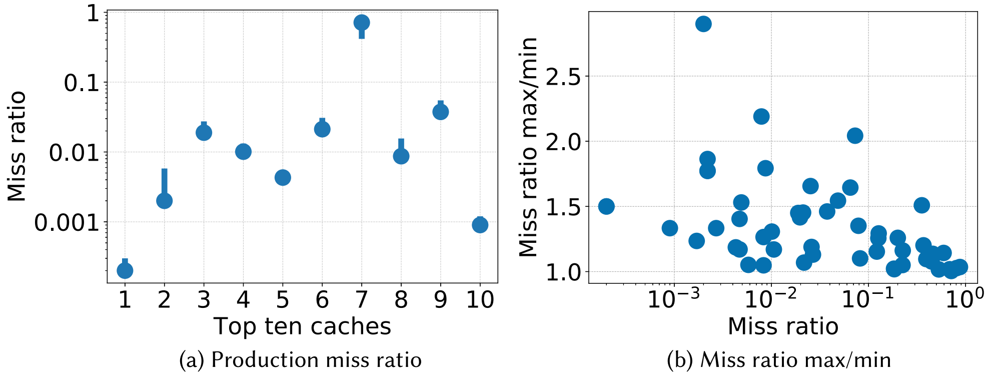
Miss ratio is one of the key metrics that indicate the effectiveness of a cache. Production in-memory caches usually operate at a low miss ratio with small miss ratio variation.

We present the miss ratios of the top ten Twemcache clusters ranked by request rates in Figure 3(a) where the dot shows the mean miss ratio over a week, and the error bars show the minimum and maximum miss ratio. Eight of the ten Twemcache clusters have a miss ratio lower than 5%, and six of them have a miss ratio close to or lower than 1%. The only exception is a write-heavy cache cluster, which has a miss ratio of around 70% (see Section 4.4.2 for details about write-heavy workloads). Compared to CDN caching [65], in-memory caching usually has a lower miss ratio.

We use miss ratio as metric instead of hit ratio, because we believe miss ratio is better suited for in-memory caching given their low miss ratios. Moreover, miss ratio more directly connects to the backend load, thus can be more useful. For example, reducing a hit ratio from 99.9% to 99.0% seems innocuous, because it is only a 0.9% change. However, the miss ratio and the number of requests hitting the backend are 10× higher.

Besides a low miss ratio, miss ratio stability is also very important. In production, it is the highest miss ratio (and request rate) that decides the QPS requirement of the backend. Therefore, a cache with a low miss ratio most of the time, but sometimes a high miss ratio is less useful than a cache with a slightly higher but more stable miss ratio. Figure 3(b) shows the ratios of $\frac{mr_{max}}{mr_{min}}$ over the course of a week for different caches, where *mr* stands for miss ratio. We observe that most caches have this ratio lower than 1.5. In addition, the caches that have larger ratios usually have a very low miss ratio.

Low miss ratios and high stability, in general, illustrate the effectiveness of production caches. However, extremely low miss ratios tend to be less robust, which means the corresponding backends have to be provisioned with more margins. Moreover, cache maintenance and failures become a major source of disruption for caches with extremely low miss ratios. The combination of these



(a) Production miss ratio

(b) Miss ratio max/min

Fig. 3. (a) Production miss ratio of the top ten Twemcache clusters ranked by request rates, the bar shows the max and min miss ratio across one week. Note that the Y-axis is in log scale. (b) The ratio between max and min miss ratio is small for most cases.

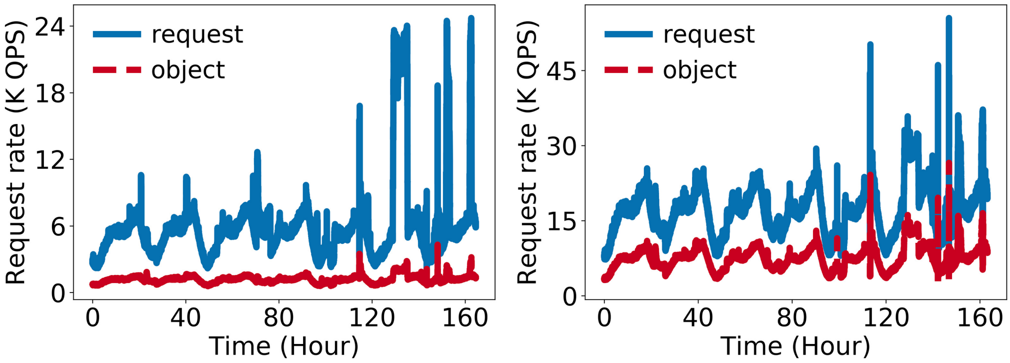


Fig. 4. The number of requests and objects being accessed every second for two cache instances from two different clusters.

factors indicate there is typically a limit to how much cache can reduce read traffic or how little traffic backends need to provision for.

4.2 Request Rate and Hot Keys

Similar to previously observed [28], request rates show diurnal patterns (Figure 4). Besides, spikes in request rate are also very common, because cache is the first responder to any change from the frontend services and end users.

When a request rate spike happens, a common belief is that hot keys cause the spikes [41, 44, 60, 66]. Indeed, load spikes often are the results of hot keys. However, we notice it is not always true. As shown in Figure 4, at times, when the request rate (top blue curve) spikes, the number of objects accessed in the same time interval (bottom red curve) also has a spike, indicating that the spikes are triggered by factors other than hot keys. Such factors include client retry requests, external traffic surges, scan-like accesses, and periodic tasks.

In addition to request rate spikes, caches often show other irregularities. For example, in Section 4.7.2, we show that it is common to see sudden changes in object size distribution. These irregularities can happen for various reasons. For instance, users change their behavior due to a social event, the frontend service adds a new feature (or bug), or an internal load test is started.

Table 1. The Cache Fanout Size Distribution in 24 Hours for Two Critical Twitter Services

Service	Number of Samples	P50	P75	P90	P99	P999	P9999	max
A	3,624,157	1	3	21	79	351	504	504
B	423,711,161	1	1	6	10	29	63	493

As a critical component in the infrastructure, caches stop most of the requests from hitting the backend, and they should be designed to tolerate these workload changes to absorb the impact.

4.3 Fanout and Tail Latency

A low tail latency is critical for an in-memory key-value cache, because client requests in modern web services usually have large fanouts [84]: One request triggers hundreds of requests to the internal services. Moreover, multiple levels of fanouts are also common in our observations: One fanout request further triggers hundreds of fanout requests to other services.

By looking into distributed tracing data, we examined the fanout distribution (number of fanout requests at different percentile) over the course of 24 h against cache used by two critical services at Twitter (Table 1). Service A serves tweet replies and service B serves user profiles. We remark that additional fanout could have happened before these requests reached the service under study. Table 1 shows that majority of the requests from these two services have a fanout size of 1. However, the tail is much larger. For example, the max fanout size from service A is over 500, and this number for B is huge as well, despite a single-digit fanout size at P75 for both.

Serving an user-facing API request triggers a cascading tree of fanout requests involving many internal services. Most high-throughput services are at least partially backed by in-memory caches. Therefore, cache tail latency is important to service scalability [50], user experience, and revenue [97].

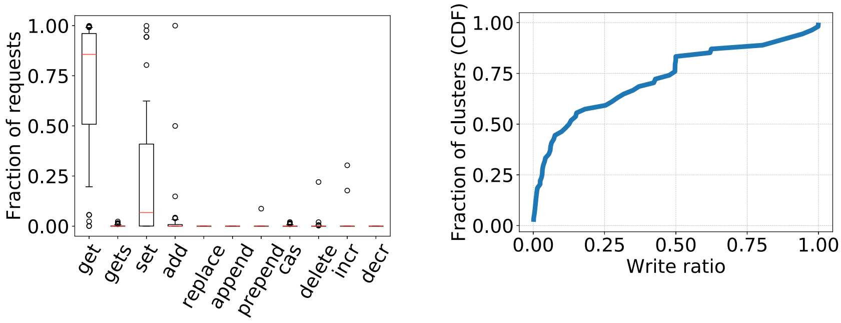
4.4 Types of Operations

Twemcache supports 11 different operations, of which `get` and `set` are the most heavily used by far. In addition, write-heavy cache workloads are very common at Twitter.

4.4.1 Relative Usage Comparison. We begin from the operations used by Twemcache workloads. Twemcache supports eleven operations `get`, `gets`, `set`, `add`, `cas` (check-and-set), `replace`, `append`, `prepend`, `delete`, `incr`, and `decr`.[2] As shown in Figure 5(a), `get` and `set` are the two most common operations, and average `get` ratio is close to 90% indicating most of the caches are serving read-heavy workloads. Apart from `get` and `set`, operations `gets`, `add`, `cas`, `delete`, and `incr` are also frequently used in Twemcache clusters. However, compared to `get` and `set`, these operations usually account for a smaller percentage of all requests.

Among the rest of operations, `add` is used more often than others because of two reasons. First, for concurrent writes, `add` operation is often preferred over `gets` + `cas`. `gets+cas` is the default approach for atomically inserting or updating a *mutable* object. However, they incur two RPC calls, which are expensive. In some cases where data race is not often, engineers use `add` to insert first. If `add` fails, then `gets+cas` are used for an atomic update. Second, `add` operation can be used to implement distributed locking service. `add` operation fails when the key being added is in the cache, so a successful `add` indicates a successful lock; otherwise, it indicates others have already obtained the lock. After obtaining a lock, the application can unlock it by removing the key from the cache. Among the rest of the operations, we noticed that `replace` is never used, because `set`

[2] See https://github.com/memcached/memcached/wiki/Commands for details about each command.



(a) Relative use of each operation

(b) Write ratio

Fig. 5. (a) Ratio of operation in each Twemcache cluster, box shows the 25th and 75th percentile, red bar inside the box shows the mean ratio, and whiskers are 10th and 90th percentile. (b) Write ratio distribution CDF across Twemcache clusters.

has a similar function: update the value when the object is in the cache, and `replace` is only needed when a user wants to guarantee the write to be an overwrite. Such a usage scenario is very rare. Operation `prepend` and `append` are rarely used, because the implementation for list operation in Twemcache and Memcached is not efficient, and engineers usually prefer Redis for such rich data structure usages. Operation `incr` is often used by counters, in which scenario, `decr` is not needed. Nevertheless, these operations serve important roles in in-memory caching. Therefore, as suggested by the maintainer of Memcached, they should not be ignored [16].

Based on the observations, it is possible to optimize existing in-memory caching systems without loss of compatibility. Here, we give an example of such optimization. Operation `gets` and `cas` are the two operations used for read-modify-write. To prevent race condition, `gets` returns a unique identifier (`cas` value) together with the object. During an update operation, cache checks whether the `cas` value has modified since last `gets`. If it has been modified, then `cas` fails and the client needs to retry. To support `cas`, Twemcache (and also Memcached) associate 8-byte version number with each stored object, which is a large space overhead compared to the sizes of objects stored in Twemcache (Section 4.7). Since the request rate of `cas` is low, one possible tradeoff is to support `cas` via other approaches, such as using a global version hash table, so that we can reduce the per-object metadata by 8 bytes.

4.4.2 Write Ratio. Although most caches are read dominant, Figure 5(a) shows that both `get` and `set` ratios have a large range across caches. We define a workload as write-heavy if the percentage sum of `set`, `add`, `cas`, `replace`, `append`, `prepend`, `incr`, and `decr` operations exceeds 30%. Figure 5(b) shows the distribution of write ratio across caches. More than 35% of all Twemcache clusters are write-heavy, and more than 20% have a write ratio higher than 50%. In other words, in addition to the well-known use case of serving read-heavy workloads, a substantial number of Twemcache clusters are used to serve write-heavy workloads. We identify the main use cases of write-heavy caches below.

Frequently updated data. Caches under this category mostly belong to cache for computation or transient data (Sections 2.4.2 and 2.4.3). Updates are accumulated in cache before the keys get accessed, or the keys eventually expire.

Opportunistic pre-computation. Some services continuously generate data for potential consumption by itself or other services. One example is the caches storing recent user activities, and the cached data are read when a query asks for recent events from a particular user. Many services choose not to fetch relevant data on demand, but instead opportunistically pre-compute them for a much larger set of users. This is feasible, because pre-computation often has a bounded cost, and in exchange read queries can be quickly fulfilled by pre-computed results partially or completely. Since this is a tradeoff mainly for user experience, the caches under this category see objects with fewer reuse. Therefore, the write ratio is often higher (>80%), and object access frequency (the number of read and write requests) is often lower. In one case, we saw one cluster with a mean object frequency close to 1.

4.5 TTL

Two important features that distinguish in-memory key-value caches from a durable key-value store are TTL and cache eviction. While evictions have been widely studied [30, 35], TTL is often overlooked. Nonetheless, TTL has been routinely used in production. Moreover, as a response to GDPR [5], the usage of caching TTL has become mandatory at Twitter to enforce data retention policies. TTL is set when an object is first created in Twemcache, and decides its expiration time. Request attempts to access an expired object will be treated as misses, so keeping expired objects in the cache is not useful.

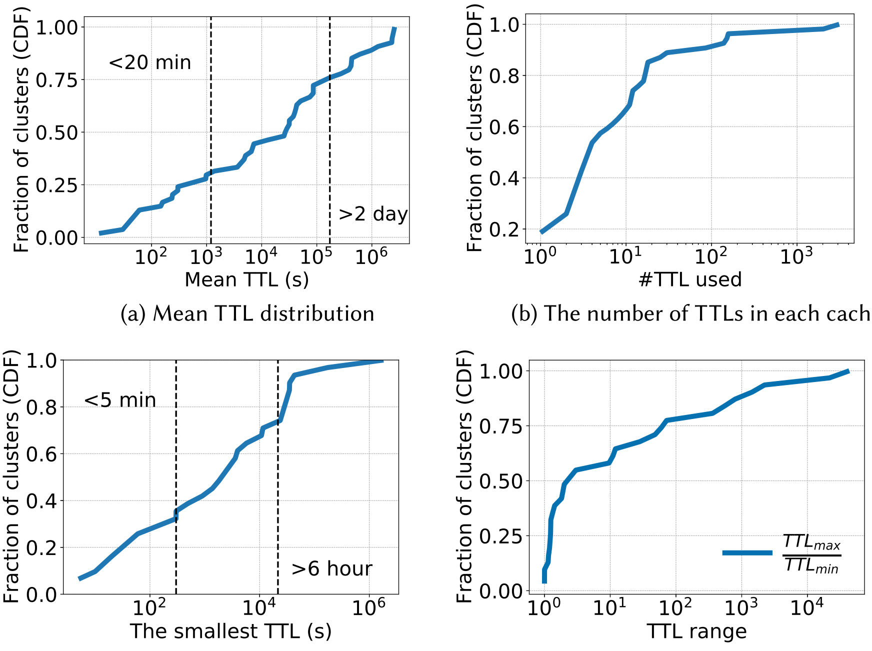
We observe that in-memory caching workloads often use short TTLs. This usage comes from the dynamic nature of cached objects and the usage for implicit deletion. Under this condition, effectively and efficiently removing expired objects from the cache becomes necessary and important, which provides an alternative (but not a replacement) to better eviction algorithm in achieving low miss ratios.

4.5.1 TTL Usages. We measure the mean TTLs used in each Twemcache cluster and show the TTL distribution in Figure 6(a), which shows that TTL ranges from minutes to days. More than 25% of the workloads use a mean TTL shorter than 20 min, and less than 25% of the workloads have a mean TTL longer than two days. Such a TTL range is longer than DNS caching (minutes) [71] but shorter than common CDN object caching (days to weeks). We divide caches into *short-TTL caches* (TTL $\leq$ 12 h) and *long-TTL caches* (TTL > 12 h). Figure 6(a) shows 66% of all Twemcache clusters have a short mean TTL.

In addition to mean TTL distribution, we have also measured the number of TTL used in each cache. Figure 6(b) shows that only 20% of the Twemcache workloads use a single TTL, while the rest majority use more than one TTL. In addition, we observe that over 30% of the workloads use more than ten TTLs and there are a few workloads using more than 1000 TTLs. In the last case, some clients intentionally scatter TTLs over a pre-defined time range to avoid objects expiring at the same time. This technique is called *TTL jitter*. In another case, the clients seek the opposite effect—computing TTLs so that a group of objects will expire at the same, predetermined time.

Besides the number of TTLs used, the smallest TTL and the TTL range, defined as the ratio between TTL_{max} and TTL_{min}, are also important for designing algorithms that remove expired objects (see Section 8). Figure 6(c) shows that the smallest TTL in each cache varies from tens of seconds to more than half day. In detail, around 30 to 35% of the caches have their smallest TTL shorter than 300 s, and over 25% of caches have the smallest TTL longer than 6 h. Figure 6(d) shows the CDF of each workload's TTL range. We observe that fewer than 40% of the workloads have a relatively small TTL range (< 2× difference), while almost 25% of the caches have $\frac{TTL_{max}}{TTL_{min}}$ over 100. Below, we present the three main purposes of TTL to better explain how TTL settings relate to the usages of the caches.

Bounding inconsistency. Objects stored in Twemcache can be highly dynamic. Because cache updates are best-effort, and failed cache writes are not always retried, it is possible that objects stored in in-memory cache are stale. Therefore, applications often use TTL to bound inconsistency,



(a) Mean TTL distribution

(b) The number of TTLs in each cache

(c) The smallest TTL distribution

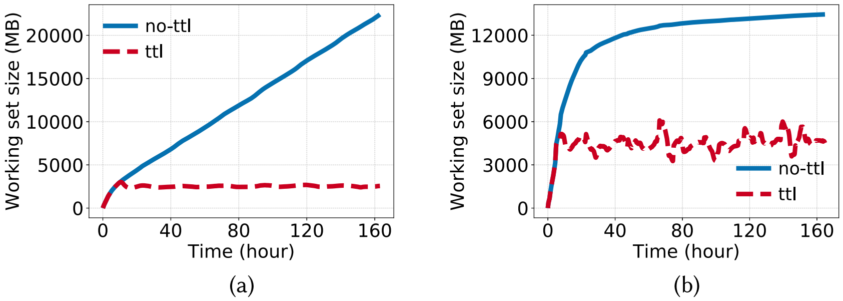
(d) TTL range distribution

Fig. 6. (a) More than half of caches have mean TTL shorter than one day. (b) Only 20% of caches use single TTL. (c) The smallest TTL in each cache can be very long. (d) TTLs ranges in workloads are often large.

which is also suggested in the AWS Redis documentation [7]. TTLs for this purpose usually have relative large values, in the range of days. Some Twitter services further developed *soft TTL* to achieve a better tradeoff between data consistency and availability. The main idea of soft TTL is to store an additional, often shorter TTL as part of the object value. When application decodes the value of a cached object and notices that the soft TTL has expired, it will refresh the cached value from its corresponding source of truth in the background. Meanwhile, the application continues to use the older value to fulfill current requests without waiting. Soft TTL is typically designed to increase with each background refresh, based on the assumption that newly created objects are more likely to see high volume of updates and therefore inconsistency.

Implicit deletion. In some caches, TTL reflects the intrinsic life span of stored objects. One example is the counters used for API rate limiting, which are declared as maximum number of requests allowed in a time window. These counters are typically stored in cache only, and their TTLs match the time windows declared in the API specification. In addition to rate limiters, TTL for GDPR compliance would also fall into this category. To comply with GDPR, no data would live in cache beyond the duration permitted by the law.

Periodic refresh. TTL is also used to promote data freshness. For example, a service that calculates how much a user's interest matches a cluster/community using ML models can make "who-to-follow" type of recommendations with the results. The results are cached for a while, because user and cluster characteristics tend to be stable in the very short term, and the calculation is relatively expensive. Nonetheless, as users engage with the site, their portraits can change over time. Therefore such a service tends to recompute the results for each user periodically, using or adding the latest data since last update. In this case, TTL is used to pace a relatively expensive operation that should only be performed infrequently. The exact value of the TTL is the result



(a)

(b)

Fig. 7. The working set size grows over time when TTL is not considered. However, when TTL is considered, the working set size is capped.

of a balance between computational resources and data freshness, and can often be dynamically updated based on circumstances.

4.5.2 Working Set Size and TTL. Having the majority of caches using short TTLs indicate that the *effective working set size* (WSS_E)—the size of all unexpired objects may be loosely bounded. In contrast, the *total working set size* (WSS_T), the size of all active objects regardless of TTL, can be unbounded.

In our measurements, we identify two types of workloads shown in Figure 7. The first type (Figure 7(a)) has a continuously growing WSS_T, and it is usually related to user-generated content. With new content being generated every second, the total working set size keeps growing. The second type of workload has a large growth rate in WSS_T at first, and then the growth rate decreases after this initial fast-growing period, as shown in Figure 7(b). This type of workloads can be users related, the first quick increase corresponds to the most active users, the slow down corresponds to less active users. Although the two workloads show different growth patterns in total working set size, the effective working set size of both arrive at a plateau after reaching its TTL. Although the WSS_E may fluctuate and grow in the long term, the growth rate is much slower compared to WSS_T.

Bounded WSS_E means that, for many caches, there exists a cache size that the cache can achieve compulsory miss ratio, if an in-memory caching system can remove expired objects in time. This suggest the importance of quickly removing expired object from cache, especially for workloads using short TTLs. Unfortunately, while eviction has been widely studied [30, 35, 78], expiration has received little attention. And we will show in Section 8.2, existing solutions fall short on expiration.

4.6 Popularity Distribution

4.6.1 Zipfian Distribution. Object popularity is another important characteristic of a caching workload. Popularity distribution is often used to describe the cachebility of a workload. A popular assumption is that cache workloads follow Zipfian distribution [36], and the frequency-rank curve plotted in the log-log scale is linear. Frequency is defined as the number of requests for an object, and rank is defined as the position in the object sequence ordered by frequency, i.e., the most popular object has the highest frequency with rank 1. A large body of work optimizes system performance under Zipfian popularity assumption [41, 51, 60, 70, 81, 87]. However, a recent work from Facebook [32] suggested that caching workloads may not follow Zipfian distribution. Here, we present the popularity of the caching workloads at Twitter.

Measuring all Twemcache workloads, we observe majority of the cache workloads still follow Zipfian distribution. However, some workloads show deviations in two ways. First, unpopular objects appear significantly less than expected (Figure 8(a)) or the most popular objects are less

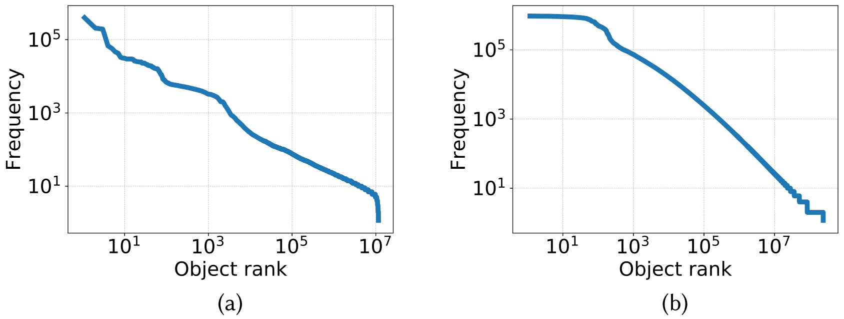


Fig. 8. Some workloads showing small deviations from Zipfian popularity. (a) The least popular objects are less popular than expected. (b) The most popular objects are less popular than expected.

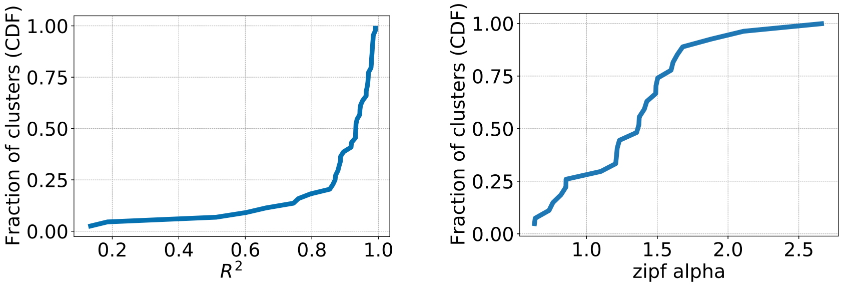
popular than expected (Figure 8(b)). The first deviation happens when objects are always accessed multiple times so that there are few objects with frequency smaller than some value. The second deviation happens when the client has an aggressive client-side caching strategy so that the most popular objects are often cached at client. In this case, the cache is no longer single-layer.

Although these deviations happen, they are rare, and we believe it is still reasonable to assume in-memory caching workloads follow Zipfian distribution. Since most part of the frequency-rank curves are linear in the log-log scale, we use linear fitting[3] confidence R^2 [12] as the metric for measuring the goodness of fit. Figure 9(a) shows the results of fitting. 80% of all workloads have R^2 larger than 0.8, and more than 50% of workloads have R^2 larger than 0.9. These results indicate that the popularity of most in-memory caching workloads at Twitter follows Zipfian distribution. We further measure the parameter α of the Zipfian distribution shown in Figure 9(b). The figure shows that most of the α values are in the range from 1 to 2.5, indicating the workloads are highly skewed. Compared to the popularity distribution of Facebook in-memory cache workloads presented in Reference [32], we observe that workloads at Twitter are more skewed. One reason could be arising from the fact that Twitter uses a public follower-type social network, where a Tweet published by a celebrity is seen (requested) by all the followers. In comparison, the core emphasis of the social network at Facebook is private mutual relationships where any content published by a user can only be seen by its friends. In addition, the social graph workloads studied in Reference [32] came from second layer caches and were highly sampled, which might also be contributing to the observed differences.

4.6.2 Count-frequency Curve. While most workloads show Zipfian distributions, the relatively small R^2 suggests the fitting is not perfect. This is potentially because the fitting for heads and tails of the curves are not very accurate. The tails of the curves show objects with only a few (or one) request(s). However, the low frequency can be caused by factors other than the properties inherent to the workloads. For example, these objects may be newly created, in which case information about these objects would be insufficient. Therefore, these objects may be popular objects that falsely show up in the tail.

Besides, the curve head can also be inaccurate in some cases. Because cache traces are usually *spatial* sampled, some of the most popular objects may not be included in the collected traces. In our case, although we did not sample the request streams in time, each trace is collected from one

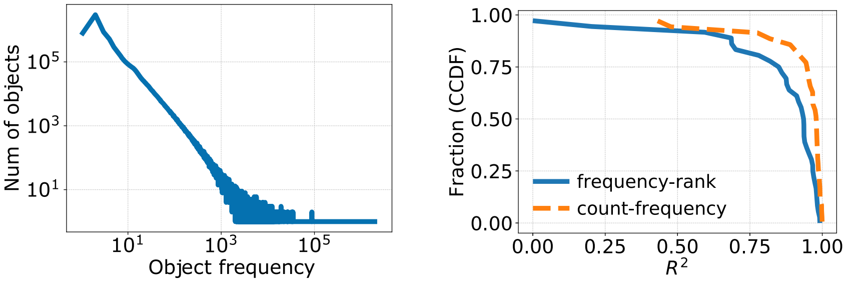
[3] We remark that linear regression is not the correct way to modeling Zipfian distribution from the view of statistics, we perform this to align with existing works [36].



(a) R^2 in Zipfian fitting

(b) Zipfian parameter α

Fig. 9. (a) Most of workloads follow Zipfian popularity distribution with large confidence R^2. (b) The parameter α in Zipfian distribution is large, and the popularity of most workloads are highly skewed ($\alpha > 1$).



(a)

(b)

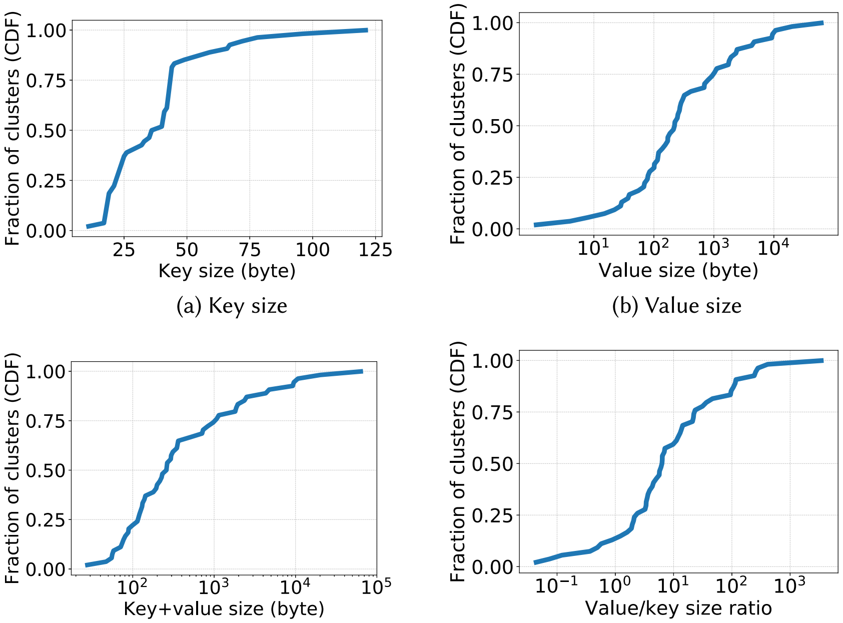
Fig. 10. (a) Typical count-frequency popularity plot. (b) R^2 distribution of linear fitting count-frequency curves.

instance of the cluster. In other words, they are spatial sampled, and each trace only contains one partition of the object space.

Inaccurate head and tail cause inaccuracies in Zipfian frequency-rank popularity modeling. To address this problem, we propose to model the object count and frequency to understand the object popularity in the workload. Figure 10(a) shows the number of objects (Y) being requested X times (frequency of the object) in the trace. We observe that when plotting this count-frequency curve on a log-log scale, the resulting curve is more closer to being linear than modeling with frequency-rank. This is because this count-frequency modeling is equivalent to frequency-rank modeling. A count-frequency curve can be converted to frequency-rank curve, and vice versa. Modeling with a count-frequency curve provides higher modeling confidence, because low-frequency (low-fidelity) objects are aggregated into a few data points and thus contribute less to the linear fitting. We perform a linear fitting of the curves, and show the R^2 in Figure 10(b). We observe that 81% of the workloads modeled with count-frequency show $R^2 > 0.9$, while only 55% for frequency-rank modeling as a comparison. The higher R^2 suggests that modeling non-strict Zipfian workloads using count-frequency might be better than frequency-rank methods. This workload modeling aspect needs further exploration, which would be an interesting future work.

4.7 Object Size

One feature that distinguishes in-memory caching from other types of caching is the object size distribution. We observe that similar to previous observations [28], the majority of objects stored in Twemcache are small. In addition, size distribution is not static over time, and both periodic distribution shifts and sudden changes are observed in multiple workloads.



(a) Key size

(b) Value size

(c) Object size

(d) Value/key size ratio

Fig. 11. Mean key, value, object size distribution, and mean $\frac{value}{key}$ size ratio across all caches.

4.7.1 Size Distribution. We measure the mean key size and value size in each Twemcache cluster and present the CDF of the distributions in Figure 11. Figure 11(a) shows that around 85% of Twemcache clusters have a mean key size smaller than 50 bytes, with a median smaller than 38 bytes. Figure 11(b) shows that the mean value size falls in the range from 10 bytes to 10 KB, and 25% of workloads show value size smaller than 100 bytes, and median is around 230 bytes. Figure 11(c) shows that CDF distribution of the mean object size (key+value), which is very close to the value size distribution except at small sizes. Value size distribution starts at size 1, while object size distribution starts from size 16. This indicates that for some of the caches, value size is dramatically smaller than the key size. Figure 11(d) shows the ratio of mean value and key sizes of each cache cluster. We observe that 15% of workloads have the mean value size smaller than or equal to the mean key size, and 50% of workloads have value size smaller than 5× key size.

4.7.2 Size Distribution Over Time. In the previous section, we investigated the static size distribution of all objects accessed in the one week's time of each Twemcache cluster. However, the object size distribution of workloads are usually not static over time. In Figure 12, we show how the size distribution changes over time. The X-axis shows the time, and the Y-axis shows the size of objects (using slab class size as bins), the color shows how much of the objects in one time window fall into each slab class. We observe that some of the workloads show diurnal patterns (Figures 12(a) and 12(b)), while others show changes without strict patterns.

Periodic/diurnal object size shifts can come from the following sources: (a) value for the same key grows over time, and (b) size distribution correlates with temporal aspects of key access. For example, text content generated by users in Japan are shorter/smaller than those by users in Germany. In this case, it is the geographical locality that drives the temporal pattern. However, we do not yet have a good understanding of how most sudden, non-recurring changes happen. Current guesses include user behavior changes during events, and a temporary change in production settings.

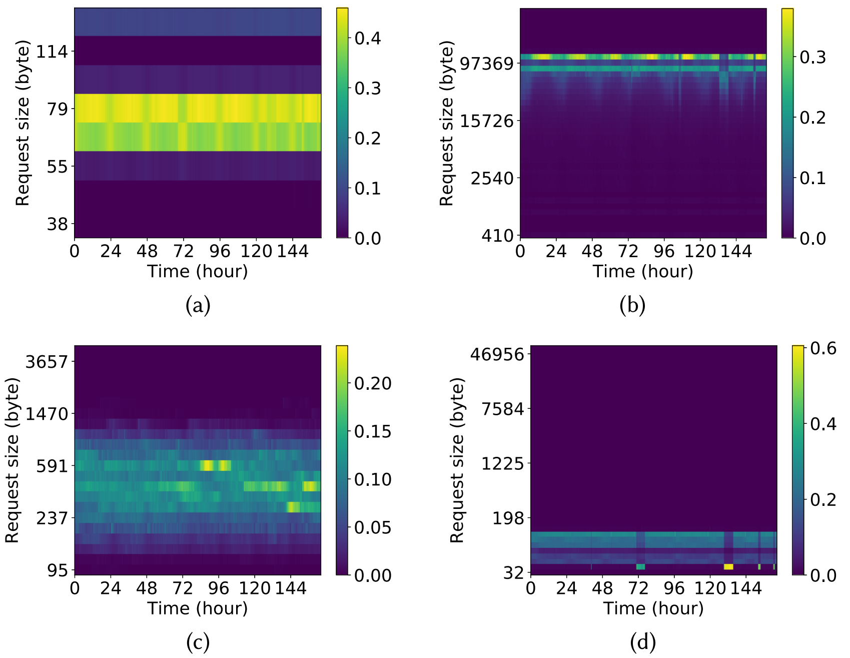


Fig. 12. Heatmap showing request size distribution over time for four typical caches. X-axis is time, Y-axis is the object size using slab class size as bins, and the color shows the fraction of requests that fall into a slab class in that time window.

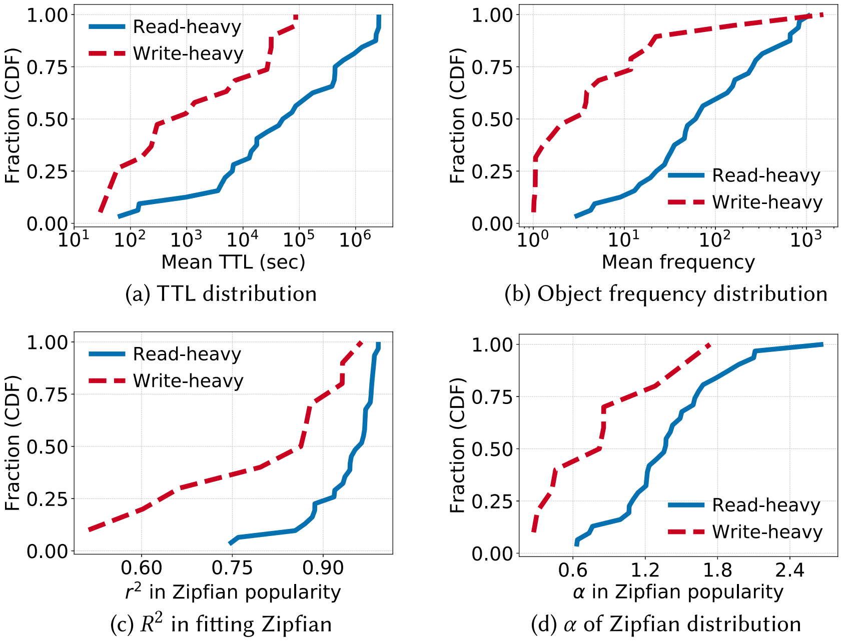
Both short-term and long-term size distribution shifts pose additional challenges to memory management in caching systems [92]. And the changes make it hard to control or predict external fragmentation in caches that use external heap memory allocators, such as Redis. While for slab-based application heap allocator such as the one in Memcached, it can cause slab calcification [111]. In Section 8.5, we discuss why existing techniques do not completely address the problem.

5 CORRELATIONS BETWEEN PROPERTIES

Throughout the analysis in previous sections, we observe some workload characteristics have strong correlations with the write ratio. For example, write-heavy workloads usually use short TTLs. Presented in Figure 13(a), the dashed red curve shows the mean TTL distribution of write-heavy workloads, and the solid blue curve shows the mean TTL distribution of read-heavy workloads. Around 50% of the write-heavy workloads have mean TTL shorter than 10 minutes, while for read-heavy workloads, this is 15 h. Further, the Pearson coefficient between write ratio and log[4] of mean TTL (Table 2) is −0.63, indicating a negative correlation, confirming that large write ratio workloads usually have short TTLs.

Besides TTL, write-heavy workloads also show low object frequencies. We present the mean object frequency (in terms of the number of accesses in the traces) of read-heavy and write-heavy workloads in Figure 13(b). It shows that read-heavy workloads have a mean frequency mostly in the range from 6 to 1,000, with 75% percentile above 200. Meanwhile, write-heavy workloads have a mean frequency mostly between 1 and 100, with 75% percentile below 10. We further confirm

[4] We choose to use log of TTL and frequency because of their wide ranges in different workloads.



(a) TTL distribution

(b) Object frequency distribution

(c) R^2 in fitting Zipfian

(d) α of Zipfian distribution

Fig. 13. Write-heavy workloads tend to show short TTL, small object access frequency, relatively large deviations from Zipfian popularity distribution, and they are usually less skewed (small α).

Table 2. Correlation between Write Ratio and Other Properties

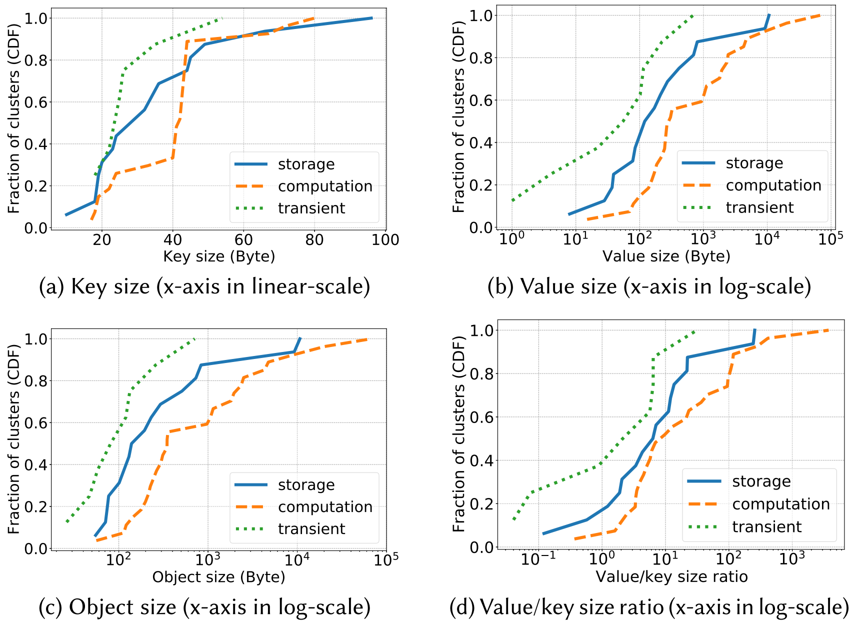
Property	Pearson coefficient with write ratio
log(TTL)	−0.6336
log(Frequency)	−0.7414
Zipfian fitting R^2	−0.7690
Zipfian alpha	−0.7329

this relationship with the Pearson coefficient between write ratio and log of frequency, which is −0.7414 (Table 2), suggesting the low object access frequency in write-heavy caches.

In addition, the popularity of write-heavy workloads has relatively larger deviations from Zipfian distribution, and the fitting confidence R^2 is usually much smaller than that of read-heavy workloads (Figure 13(c)). Moreover, the α parameter of Zipfian distribution in write-heavy workloads is usually small, as shown in Figure 13(d). It shows the write-heavy workloads have a median α around 0.9, and the median of read-heavy workloads have an α around 1.4. This correlation is also backed up by the Pearson coefficient (Table 2).

6 PROPERTIES OF DIFFERENT CACHE USE CASES

Here, we further explore common properties exhibited by each of the three major caching use cases as described in Section 2.4. We show the size distribution curves of the caches under each use case in Figure 14, write ratio, mean TTL, and popularity alpha distribution in Figure 15.



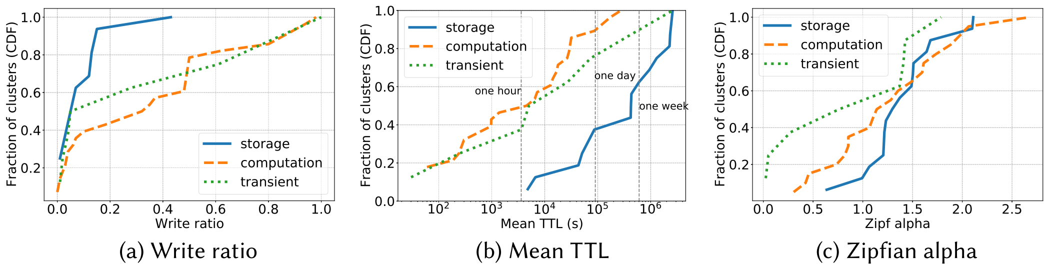
(a) Key size (x-axis in linear-scale)

(b) Value size (x-axis in log-scale)

(c) Object size (x-axis in log-scale)

(d) Value/key size ratio (x-axis in log-scale)

Fig. 14. The mean key, value, object size distribution, and mean $\frac{value}{key}$ size ratio across all caches of each use case.



(a) Write ratio

(b) Mean TTL

(c) Zipfian alpha

Fig. 15. Write ratio, mean TTL, and popularity Zipfian alpha distribution for each cache use case.

6.1 Caches for Storage

Caches for storage represents the most typical in-memory cache workload, whose properties follow the observations in previous works [28, 84]. Figure 14(a) shows that caches under this use case have key sizes in the range of a few to around 100 bytes. Figure 14(b) shows that value sizes are mostly in the range of 10s of bytes to under 1 KB. The object size curve for storage (Figure 14(c)) is similar to the value size curve, and the median is slightly larger than 100 bytes, indicating that object sizes are small in these caches. Compared to value size, 20% of the caches have an equal or larger key size, and 60% of the caches have the key size in the same order of magnitude as value size (Figure 14(d)).

As the most typical in-memory caches, caches for storage mostly serve ready-heavy workloads (Figure 15(a)) with write ratio ranging from close to 0 to 42%. Further, we observe that majority of the workloads have write ratios smaller than 10%. These caches typically follow the Zipfian distribution and show a large parameter α from 0.6 to 2.2 with a majority between 1 and 1.6. Because this type of workload is highly skewed, they are easier to cache, and in production, 95% of these clusters have miss ratios of around or less than 1%. Being more cacheable and having smaller miss ratios do not indicate they have small working set sizes. In our observation, 7 of the top 10 caches (ranked by cache size) belong to this category.

Because caches in this category cache objects persisted in the backend storage, modifications to these objects are explicitly written to both the backend and the cache. As a result, the TTLs used in these caches are usually longer than others, from a few hours to more than one week, with most of the caches using TTLs longer than one day.

Compared to the other two use cases, smaller write ratios and longer TTLs are the most prominent characteristics of these caches.

6.2 Caches for Computation

Compared to caches for storage, caches for computation usually have larger key and value sizes. Figure 14(c) shows that the median object size is around 300 bytes, almost 3× larger than caches for storage. Moreover, 75 percentile object size is more than 6× larger than the caches for storage. Because object value sizes are larger for these caches, the value-over-key-size ratios tend to be large as well (Figure 14(d)).

Caches under this category serve both read-heavy and write-heavy traffic. We observe that the median write ratio is around 35%, and 60% of the caches show write ratios smaller than 50%. The mix of ready-heavy and write-heavy workloads comes from the diverse usages in this class of caches. For example, machine learning features and predicted results are usually read-heavy, showing a good fit of Zipfian popularity distribution, which are similar to caches for storage in access patterns. In contrast, intermediate computation results and pre-computation (Section 4.4.2) results are often write-heavy and show deviations from Zipfian distribution. Such deviation is because these results are typically read at most a few times and sometimes have no read before the next write (update).

Compared to caches for storage, workloads under this category use shorter TTLs. Figure 15(b) shows that the median TTL is around 2 h. As a comparison, the median TTL of caches for storage is six days. Such short TTLs are usually determined by the application requirement. For example, caches storing intermediate computation data usually have TTLs no more than a few minutes, because other services typically consume the data soon after being produced. The TTLs of features and prediction results are usually from minutes to hours (and some up to days), depending on how fast the underlying data change and how expensive the computation is.

Since objects stored in these caches are indirectly related to users and contents, the workloads usually have large keyspaces. For example, a cache storing the distance between two users will require a N^2 cache size where N denotes the number of users. Together with larger object size, the total working set sizes are usually larger. However, because these caches use short TTLs, the effective working set sizes are usually much smaller. Thus removing expired objects in a timely fashion can be more critical than object eviction in these caches.

As real-time stream processing and machine learning become more popular in modern web applications, we envision more caches being provisioned for caching computation results. Because these caches' characteristics are different from caching for storage, they may not benefit equally from optimizations that only aim to make the read path fast and scalable, such as optimistic cuckoo hashing [59]. Therefore, including evaluation on workloads that are write-heavy and more ephemeral will paint a complete picture of a caching system's capabilities.

6.3 Transient Data with No Backing Store

Among the three cache use cases, caches storing transient data have the smallest object size with a median value size around 60 bytes and a median object size around 80 bytes. Because the values are small, the keys are relatively large for objects in these caches, and we observe the median value-over-key-size ratio around 2 (i.e., values are just two times larger than the keys). The reason is that several caches in this category are limiters in which values are integers, and negative caches in which values are Boolean values.

In addition, caches under this category show a less skewed popularity, with a median Zipfian alpha around 0.75. Except for these two properties, caches storing transient data are similar to caches for computation: having short TTLs and serving a mix of read-heavy and write-heavy workloads. We find that most caches using TTLs to enforce implicit object deletion (Section 4.5) belong to this category.

Although caches of this type only contribute to 9% of the total Twemcache cluster request rate and 8% of total cache sizes, they play an irreplaceable role in site operations.

7 EVICTION ALGORITHMS

We have shown the characteristics of in-memory cache workloads in the previous sections. In this section, we use the same cache traces to investigate the impact of eviction algorithms. This evaluation considers production algorithms offered by Twemcache and other production systems.

7.1 Eviction Algorithm Candidates

Object LRU and object FIFO. LRU and FIFO are the most common algorithms used in production caching systems [4, 21]. However, they cannot be applied to systems using slab-based memory management such as Twemcache without modification. Therefore, we evaluate LRU and FIFO assuming the workloads are served using a non-slab-based caching system, while ignoring memory inefficiency caused by external fragmentation. As a result, we expect that the results to have a bias toward the effectiveness of LRU and FIFO compared to the three slab-based algorithms. Production results for these two algorithms might be worse than what is suggested in this section, depending on the workloads.

slabLRU and slabLRC. These two algorithms are part of eviction algorithms offered in Twemcache. slabLRU and slabLRC are equivalent to LRU and FIFO but executed at a level much coarser granularity of slabs rather than a single object. Twitter employs these algorithms to alleviate the effect of slab calcification and also to reduce the size of per-object metadata.

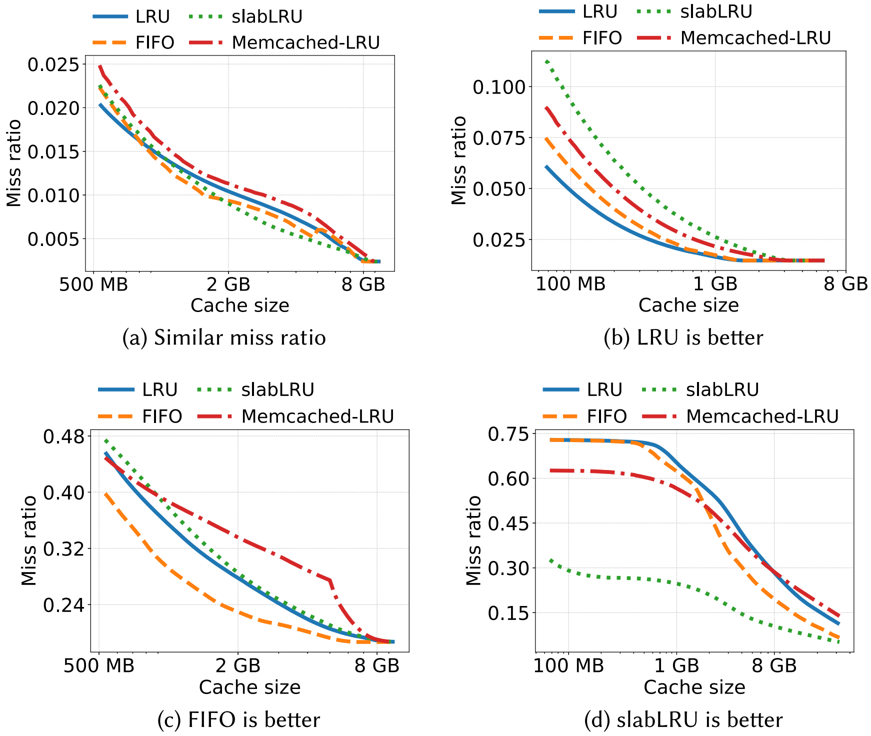
Random slab eviction. Besides slabLRU and slabLRC, Twemcache also offers Random slab eviction, which globally picks a random slab to evict. This algorithm is workload-agnostic with robust behavior, and therefore it used as the default policy in production. However, it is rarely the best of all algorithms and is non-deterministic; therefore, we do not include it in comparison.

Memcached-LRU. Memcached adapted LRU by creating one LRU queue per slab class. We call the resulted eviction algorithm Memcached-LRU, which does not enable Memcached's slab auto-move functionality. We did, however, evaluate Memcached-LRU with slab auto-move turned on, and most of the results are somewhere between LRU and slabLRU. The rest of the article omits this combination.

7.2 Simulation Setup

We built an open-source simulator called libCacheSim [108] to study the steady-state miss ratio of the different eviction algorithms. Specifically, we use five-day traces to warm up the caches, then use one-day traces to evaluate cache miss ratios. Each algorithm is applied against all traces, and then grouped by results.

In terms of cache sizes, our simulation always starts with 64 MB of DRAM and chooses the maximum as 2× their current memory in production. We stop increasing the size for a particular



(a) Similar miss ratio

(b) LRU is better

(c) FIFO is better

(d) slabLRU is better

Fig. 16. Four typical miss ratio results: (a) all algorithms have similar performance, (b) LRU is slightly better than others, (c) FIFO is better than others, and (d) slabLRU is much better than others.

workload when all algorithms have reached the compulsory miss ratio. Note that when plotting, the size range is truncated to better present the trend.

7.3 Miss Ratio Comparison

The outcome of our comparison can be grouped into four types, and representatives of each are shown in Figure 16.

The first group shows comparable miss ratios for all algorithms in the cache sizes we evaluated. For this type of workload, the choice of eviction algorithms has a limited impact on the miss ratio. Production deployments may very well favor simplicity or decide based on other operational considerations such as memory fragmentation. Twemcache uses random slab eviction by default, because random eviction is simple and requires less metadata.

The second type of result shows that for some workloads LRU works better than others. Such a result is often expected, because LRU protects recently accessed objects and is well-known for its miss ratio performance in workloads with strong temporal locality.

The third type of result shows that FIFO is the best eviction algorithm (Figure 16(c)). This result is somewhat surprising, since it does not conform to what is typically observed in caching of other scenarios such as CDN caching. We give our suspected reasons below. Figure 17 shows the inter-arrival time distribution of the two workloads in Figures 16(b) and 16(c), respectively. The inter-arrival time is the number of requests between two accesses to the same object. Figure 17(a) shows a smooth inter-arrival time curve, while Figure 17(b) shows a curve with multiple segments. For workloads with inter-arrival time like Figure 17(a), LRU can work better than FIFO, because

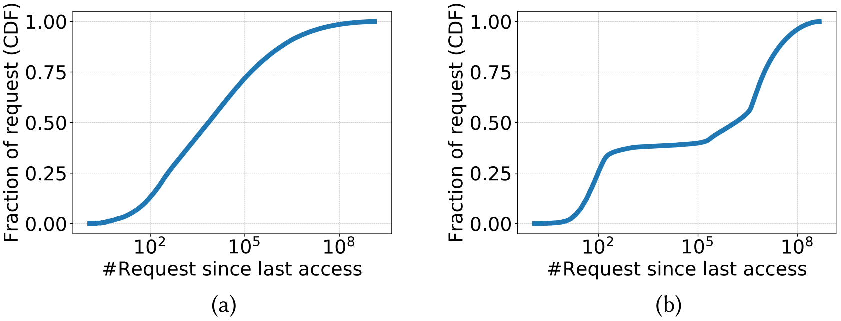


Fig. 17. The inter-arrival gap distribution corresponding to the workloads in Figures 16(b) and 16(c), respectively.

it promotes recently accessed objects, which have a higher chance of being reused soon. This promotion protects the recently accessed objects but demotes other objects that are not reused recently. Demoting non-recently used objects can be an unwise decision if some of the demoted objects will be reused after 10^6 requests, such as the ones shown in Figure 17(b). In contrast, FIFO treats each stored object equally; in other words, it protects the objects with a large inter-arrival gap. Therefore, for workloads similar to the one in Figure 17(b), FIFO can perform better than LRU. Such workloads may include scan type of requests such as a service that periodically sends emails.

The last type of result show that in some workloads, slabLRU performs much better than any other algorithms. The main reason is that the workloads showing this type of result have periodic/diurnal changes. Figure 12(b) shows the object size distribution over time of the workload corresponding to Figure 16(d). We suspect this is due to the following reason, but we leave the verification as future work. Although LRU and FIFO are not affected by any change in object size distribution, they cannot respond to workload change instantly. In contrast, slabLRU can quickly adapt to a new workload when the new workload uses a different slab class, because it prioritizes the slabs that have more recent access. From another view, slabLRU gives a larger usable cache size for the new workloads (slab class). Figure 16(d) shows that the difference between algorithms reduces at larger cache sizes, this is because the benefit of having a large usable cache size diminishes as cache size increases. Moreover, in these workloads, Memcached-LRU sometimes has better performance than LRU, but for most of the workloads, Memcached-LRU is worse (not shown in the figure) because of the missing capability of moving slabs. Thus it has a smaller usable cache size. When Memcached-LRU has better performance at small cache sizes, we suspect that the changing workloads cause thrashing for LRU and FIFO [31]. Since Memcached-LRU can only evict objects within from the same slab class as the new object, it protects the objects in other slab classes from thrashing, thus showing better performance.

In most cases, both miss ratio and the difference between algorithms decrease as cache capacity increases. We observe that within our simulation configuration, which stops at or before 2× current size, the difference between algorithms eventually disappears. This suggests that to achieve low miss ratio in real life, it can be quite effective to create implementations that increase the effective cache capacity, such as through metadata reduction, adopting higher capacity media, or data compression.

Given there are more than a couple of workloads showing each of the four result types, we set out to explore whether there is one algorithm that is often the best or close to the best most of the time.

In the next section, we explore how often each algorithm is the best with a special focus on LRU and FIFO.

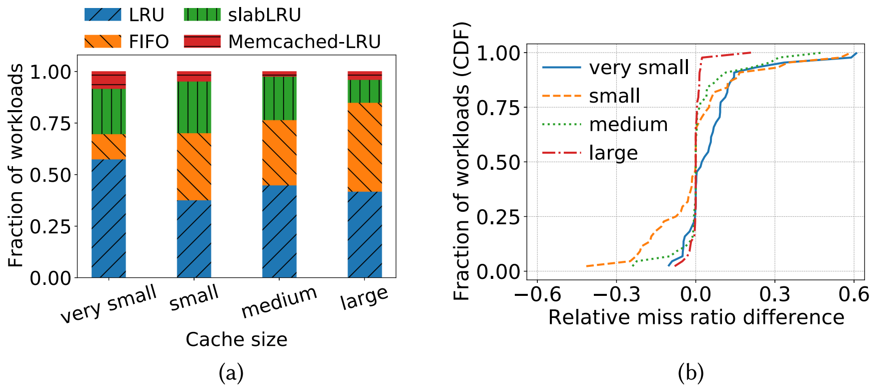


Fig. 18. (a) The best eviction algorithms under different sizes. (b) The relative miss ratio difference between FIFO and LRU under different sizes. Positive region shows FIFO is worse.

7.4 Aggregated Statistics

In this section, we evaluate the same set of algorithms as in Section 7.3, focusing on four distinct cache sizes and present the aggregated statistics. Because different workloads have different working set sizes and compulsory miss ratios, we choose the four cache sizes in the following way. We define the *ultimate cache size* s_u to be the size where LRU achieves compulsory miss ratio for a workload. However, if LRU cannot achieve compulsory miss ratio at 2× production cache size, then we use 2× production cache size as s_u. We choose *large* cache size to be 90% of s_u, and *medium*, *small* and *very small* cache sizes to be 60%, 20%, and 5% of s_u, respectively. We remark that, at Twitter, 76% of the caches have cache sizes larger than the *large* cache size category, and 34% of the rest have cache sizes within 10% of the *large* cache size.

We show the miss ratio comparison in Figure 18(a), where each bar shows the fraction of workloads for which a particular algorithm is the best. We see that at the *large* cache size slabLRU is the best for around 10% of workloads, and this fraction gradually increases as we reduce cache size. This increase is because for smaller cache sizes, quickly adapting to workload change is more valuable. Besides this, FIFO has similar performance compared to LRU at *small, medium,* and *large* size categories. And only at *very small* cache sizes, LRU becomes significantly better than FIFO. This is because at relatively large cache sizes, promoting recently accessed objects is less crucial. Instead, not demoting other objects is more helpful in improving the miss ratio, especially for workloads having multiple segments in inter-arrival time like the one shown in Figure 17(b).

Figure 18(a) suggests that for close to half of the workloads, FIFO is as good as LRU at reasonably large cache sizes. Now, we explore the magnitude by which FIFO is better or worse compared to LRU on each workload. Figure 18(b) shows the relative miss ratio difference between FIFO and LRU: $\left(\frac{mr_{\text{FIFO}}-mr_{\text{LRU}}}{mr_{\text{LRU}}}\right)$, where mr stands for miss ratio, for each workload at different cache sizes. When the value on X-axis is positive, it indicates that FIFO has a higher miss ratio, and LRU has better performance, while a negative value indicates the opposite. We observe that all the curves except the one for very small cache size are all close to being symmetric around x-axis value 0. This indicates that across workloads, FIFO and LRU have similar performance for small, medium and large cache sizes. For the very small size category, we observe LRU being significantly better than FIFO, this is because for workloads with temporal locality, promoting recently accessed objects becomes crucial at very small cache sizes. In production, most of the caches are running at cache sizes larger than or close to the *large* category. We believe that for most in-memory caching workloads, FIFO and LRU have a similar performance at reasonably large cache sizes.

The fact that FIFO and LRU often exhibit similar performance in production-like settings is important, because using LRU usually incurs extra computational and memory overhead compared to FIFO [79, 80]. For example, implementing LRU in Memcached requires extra metadata and locks, some of which can be removed if FIFO is used.

8 IMPLICATIONS

In this section, we show how our observations differ from previous work, and what the takeaways are for informing future in-memory caching research.

8.1 Write-heavy Caches

Although 70% of the top 20 Twemcache clusters serve read-heavy workloads (Section 4.4.2), write-heavy workloads are also common for in-memory caching. This is not unique to Twitter. Previous work [28] from Facebook also pointed out the existence of write-heavy workloads, although the prevalence of them were not discussed due to the limited number of workloads. Furthermore, write-heavy workloads are expected to increase in prominence as the use case of caching for computation increases (Section 2.4.2). However, most of the existing systems, optimizations, and research assume a read-heavy workload.

Write-heavy workloads in caching systems usually have lower throughput and higher latency, because the write path usually involves more work and can trigger more expensive events such as eviction. In Twitter's production, we observe that serving write-heavy workloads tend to have higher tail latencies. Scaling writes with many threads tends to be more challenging as well. In addition, as discussed in Section 5, write-heavy workloads have shorter TTLs with less skewed popularity, which are in sharp contrast to read-heavy workloads. This calls for future research on designing systems and solutions that consider performance on write-heavy workloads.

8.2 Short TTLs

In Section 4.5.1, we show that in-memory caching workloads frequently use short TTLs, and the usage of short TTLs reduces the effective working set size. Therefore, removing expired objects from the cache is far more important than evictions in some cases. In this section, we show that existing techniques for proactively removing expired objects (termed proactive expiration) are not sufficient. This calls for future work on better proactive expiration designs for in-memory caching systems.

Transient object cache. An approach employed for proactive expiration (especially for handling short TTLs), proposed in the context of in-memory caches at Facebook [84], is to use a separate memory pool (called transient object pool) to store short-lived objects. The transient object cache consists of a circular buffer of size t with the element at index i being a linked list storing objects expiring after i seconds. Every second, all objects in the first linked list expire and are removed from the cache, then all other linked lists advance by one.

This approach is effective only when the cache user uses a mix of very short and long TTLs with the short TTL usually in the range of seconds. Since objects in the transient pool are never evicted before expiration, the size of transient pool can grow unbounded and cause objects in the normal pool to be evicted. In addition, the TTL threshold of admitting into transient object pool is non-trivial to optimize.

As we show in Figure 6(b), 20% of the Twemcache workloads use a single TTL. For these workloads, transient object pool does not apply. For the workloads using multiple TTLs, we observe that fewer than 35% have their smallest TTL shorter than 300 s, and over 25% of caches have the smallest TTL longer than 6 h (Figure 6(c)). This indicates that the idea of transient object cache is not applicable to a large fraction of Twemcache clusters.

Background crawler. Another approach for proactive expiration, which is employed in Memcached, is to use a background crawler that proactively removes expired objects by scanning all stored objects.

Using a background crawler is effective when TTLs used in the cache do not have a broad range. While scanning is effective, it is not efficient. If the cache scans all the objects every T_{pass}, then an object of TTL t can be scanned up to $1 + \lceil \frac{t}{T_{pass}} \rceil$ times before removal, and can overstay in the system by up to T_{pass}. The cache operator therefore has to make a tradeoff between wasted space and the additional CPU cycles and memory bandwidth needed for scanning. This tradeoff gets harder if a cache has a wide TTL range, which is common as observed in Section 4.5. While the Twemcache workloads are single tenant, wide TTL range issue would be further exacerbated for multi-tenant caches.

Figure 6(d) shows that TTLs used within each workload have a wide range. Close to 60% of workloads have the maximum TTL more than twice as long as the minimum, and 25% of workloads show a ratio at or above 100. This indicates that for the 25% of caches, if we want to ensure all objects are removed within 2× their TTLs, objects with the longest TTL will be scanned 100 times before expiration.

The combination of transient object cache with background crawler could extend the coverage of workloads that can be efficiently expired. However, the tradeoff between wasted space and the additional CPU cycles and memory bandwidth consumed for scanning would still remain. Hence, future innovation is necessary to fundamentally address use cases where TTLs exhibit a broad range.

8.3 Highly Skewed Object Popularity

Our work shows that the object popularity of in-memory caching can be far more skewed than previously shown [32], or compared to studies on web proxy workloads [36] and CDN workloads [65]. We suspect this has a lot to do with the nature of Twitter's product, which puts great emphasis on the timeliness of its content. It remains to be seen whether this is a widespread pattern or trend. Cache workloads are also more skewed compared to NoSQL database such as RocksDB [19], which is not surprising, because database traffic is often already filtered by caches and has the most skewed portion removed via cache hits. In other words, in-memory caching and NoSQL database often observe different traffic even for the same application. Besides these two reasons, sampling sometimes results in bias in the popularity modelling, and we avoid this by collecting unsampled traces. Our observation that the workloads still follow Zipfian distribution with large alpha value emphasizes the importance of addressing load imbalance [60, 81, 87].

8.4 Object Size

Similar to previous observation [28], we observe that objects cached in in-memory caching are often tiny (Section 4.7). As a result, in-memory caches are not always bound by memory size; instead, close to 20% of the Twemcache clusters are CPU-bound.

8.4.1 Metadata Size.

Small objects signify the relatively large overhead of metadata. Reducing object metadata can yield substantial benefits for caching tiny objects. Memcached stores 56 bytes of metadata with each object. For some workloads, the object metadata can account for more than half of the cache size. Twitter's unified cache backend Pelikan [17] addresses this problem using two different solutions. First, Pelikan eliminates two LRU chain pointers used for object-based eviction by only permitting slab-based eviction. This reduces per-object metadata to 38 bytes. Second, for workloads with near-uniform object sizes, Pelikan stores objects in a fixed sized array using cuckoo hashing. This design reduces per-object metadata size by close to 90%. However, it

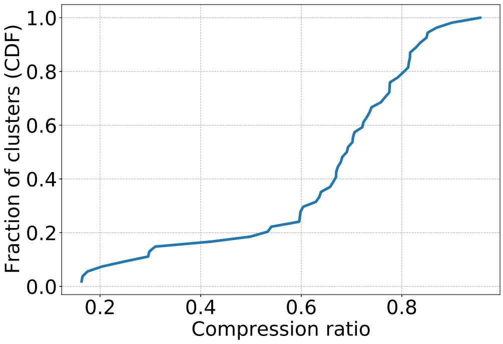


Fig. 19. Compression ratio ($1 - \frac{size_{compressed}}{size_{decompressed}}$) of keys across caches.

only benefits a limited number of workloads, because most workloads have a wide range of object sizes.

There have been several works aiming to reduce metadata size [48, 59, 80, 93]; however, they only focus on the metadata that supports get and set, and they ignore metadata that supports reference count, atomic operation (cas), and time-related operations. Several other works [30, 34, 35, 109] on reducing miss ratio (improving memory efficiency) focus on improving eviction algorithms, admission algorithms or prefetching algorithms, sometimes end up adding more metadata. In practice, reducing metadata size is an important measure to improve the effectiveness of cache.

8.4.2 Key Size. We observe that compared to value size, the key size can be enormous in some workloads. For 60% of the workloads we have studied, the mean key size and mean value size are in the same order of magnitude. This indicates that reducing key size can reduce the miss ratio for these workloads. We studied how keys are composed in the workloads and observe that a non-trivial number of workloads have namespaces as part of the object keys. For example, the key can be in the form of "namespace1 delimiter namespace2 ... delimiter id" such as "tweet:cnt:fav:12345678," which represents the favorite count of the tweet 12345678. The use of namespace is not unique to Twitter, similar observations have been made on RocksDB workloads at Facebook [39]. Such key formats help with easy debugging and are sometimes used to mirror the naming in a multi-tenant database. However, the namespaces can occupy large fractions of precious cache space while being highly repetitive.

To see the potential of key compression, we evaluated an offline method by compressing the unique keys using LZW compression [103]. Figure 19 shows the compression ratio ($1 - \frac{size_{compressed}}{size_{decompressed}}$) across cache clusters. A cache with a higher key compression ratio means more redundancy in the keys, thus a large potential for reducing miss ratio from compressing keys. The figure shows that half of the caches can reduce the key size by close to 70%, and more than 75% of the caches can reduce the key size by more than 50%. Such a large compression potential suggests one method to reduce the miss ratio: compressing keys, especially for the caches with small-sized values.

Although simple key compression is promising, there are a few challenges associated with it. First, it is difficult to identify the namespace(s) of the keys. Services use different numbers of namespaces with different namespace delimiters, and order namespaces and object id differently. Although most of the workloads use colon ":" as the delimiter, we observe that a non-trivial number of workloads use other symbols such as "/." In addition, instead of ordering the namespaces as "tweet:cnt:fav:12345678," the service might use "12345678:tweet:cnt:fav."

Second, it is tricky to perform online and robust key compression. Any compression technique designed should be robust to key pattern change and have little computation and storage overhead. One possible solution is to combine the key compression and lookup structure. For example, one can replace the hash table with a trie to achieve both compression and lookup. However, this requires a non-trivial number of optimizations to achieve a similar performance as current production systems.

Our observations call for more attention to the optimization of cache metadata and object keys. To encourage and facilitate future research on this, we keep the original but anonymized namespaces in our open sourced traces.

8.5 Dynamic Object Size Distribution

In Section 4.7.2, we show that the object size distribution is not static, and the distribution shifts over time can cause out-of-memory exceptions for caching systems using external allocators, or slab calcification for those using slab-based memory management. To solve this problem, one solution, employed by Facebook, is to migrate slabs between slab classes by balancing the age of the oldest items in each class [84]. Earlier versions of Memcached approached this problem by balancing the eviction rate of each slab class. Since version 1.6.6, Memcached has also moved to using the solution of balancing the age as mentioned above.

Besides efforts in production systems, slab assignment and migration has also been a hot topic in recent research [38, 46, 47, 64]. However, to the best of our knowledge, the problem has only been studied under a “semi-static” request sequence. Specifically, the research so far assumes that the miss ratio curve or some other properties of each slab class hold steady for =certain amount of time, which often precludes periodic and sudden changes in object size distribution.

In general, the temporal properties of object sizes in cache are not well understood or quantified. As presented in Figures 12(c) and 12(d), it is not rare to see unexpected changes in size distribution only lasting for a few hours. Sometimes it is hard to pinpoint the root cause of such changes. Nonetheless, we believe that temporal changes related to object size, whether recurring or as a one-off, usually have drivers with roots beyond the time dimension. For example, the tweet size drift throughout the day may very well depend on the locales or geo-location of active users. Some caches may be shared by datasets that differ in size distribution and access cycles, resulting in different distributions dominating the access pattern at different instants of the day. In this sense, studying the object size distribution over time could very well provide deeper insights into characteristics of the datasets being cached. Considering the increasing interest in using machine learning and other statistical tools to study and predict caching behavior, we think object size dynamics might provide a good proxy to evaluate the relationship between basic dataset attributes and their behavior in cache, allowing caching systems to make smarter decisions over time.

9 RELATED WORK

This article is an extension of our previous work [110]. Due to the nature of this work, we have discussed related works in detail throughout the article. In this section, we briefly discuss these and other works analyzing caching and storage workloads.

Multiple caching and storage system traces were collected and analyzed in the past [28, 29, 39, 65, 66, 84, 115]; however, only a limited number of reports focus on in-memory key-value caching workloads [28, 66, 84]. The closet work to our analysis is Facebook’s Memcached workload analysis [28], which examined five Memcached pools at Facebook. Similar to the observations in this work [28], we observe the sizes of objects stored in Twemcache are small, and diurnal patterns are common in multiple characteristics. After analyzing 153

Twemcache clusters at Twitter, in addition to previous observations [28], we show that write-heavy workloads are popular. Moreover, we focus on several aspects of in-memory caching that have not been studied to the best of our knowledge, including TTL and cache dynamics. Although previous work [28] proposed analytical models on the key size, value size, and inter-arrival gap distribution, the models do not fully capture all the dimensions of production caching workloads such as changing working set and dynamic object size distribution. Compared to synthetic workload models, the collection of real-world traces that we collected and open sourced provide a detailed picture of various aspects of the workloads of production in-memory caches. Besides workload analysis on Memcached, there have been several workload analysis on web proxy [25–27, 68, 96] and CDN caching [65, 104]. The photo caching and serving infrastructure at Facebook has been studied [65], with a focus on the effect of layering in caching along with the relationship between content popularity, age, and social-networking metrics.

In addition to caching in web proxies and CDNs, the effectiveness of caching is often discussed in workload studies [29, 86, 88, 100] of file systems. However, these works primarily studied the cache to the extent that of its effectiveness in reducing traffic to the storage system rather than on aspects that affect the design of the cache itself. Besides, file system caching is different from distributed in-memory caches due to a variety of reasons. For example, file system caches usually stores objects of fixed-sized chunks (512 bytes, 4 KB or larger), while in-memory caches store objects of a much wider range (Section 4.7), and scan is common in file systems, while rare in in-memory caches.

Because of the similarities in the interface, in-memory caching is sometimes discussed together with durable key-value stores. Three different RocksDB workloads [39] at Facebook has been studied in depth, with a focus on the distribution of key and value sizes, locality, and diurnal patterns in different metrics. Although Twemcache and RocksDB have a similar key-value interface, they are fundamentally different because of their design and usage. RocksDB stores data for persistence, while Twemcache stores data to provide low latency and high throughput without persistence. In addition, compared to RocksDB, TTL and evictions are unique to in-memory caching.

Besides independent workload analysis works in various caching and storage systems, some of the works on improving cache efficiency also describe observations from different production workloads. For example, Robinhood [33] describes observations on the tail latency variations over time. Adaptsize [34] shows that the optimal size for cache admission varies over time. In addition to these works, a large number of works are devoted to improving cache efficiency and performance [22, 30, 43, 45, 53, 56–58, 62, 67, 73, 77, 78, 83, 85, 91, 94, 99, 101, 102, 112, 113, 116, 117].

10 CONCLUSION

We studied the workloads of 153 in-memory cache clusters at Twitter and discovered five important facts about in-memory caching. First, although read-heavy workloads account for more than half of the resource usages, write-heavy workloads are also common. Second, in-memory caching clients often use short TTLs, which limits the effective working set size. Thus, removing expired objects needs to be prioritized before evictions. Third, read-heavy in-memory caching workloads follow Zipfian popularity distribution with a large skew. Fourth, the object size distributions of most workloads are not static. Instead, it changes over time with both diurnal patterns and sudden changes, highlighting the importance of slab migration for slab-based in-memory caching systems. Last, for a significant number of workloads, FIFO has similar or lower miss ratio performance as LRU for in-memory caching workloads. We have open sourced the traces collected at https://github.com/twitter/cache-trace.

ACKNOWLEDGMENTS

We thank our OSDI 2020 shepherd Andrea Arpaci-Dusseau, the anonymous reviewers from OSDI 2020, and TOS for their valuable feedback. We also thank Rebecca Isaacs for providing extensive feedback and helping with distributed tracing log analysis. We thank our colleague Jack Kosaian for his extensive reviews and comments that improved this work. We thank Zhexuan Li for his help in understanding the equivalence of frequency-rank curve and count-frequency curve. We also thank the Cache team and IOP team at Twitter for their support in collecting and analyzing the traces, and Daniel Berger for his comments in the early stage of the project. Moreover, we thank CloudLab [52] in providing testbed, and Geoff Kuenning from SNIA for his help hosting and sharing the traces.

REFERENCES

[1] [n.d.]. Anonymized Twitter Production Cache Traces. Retrieved from https://github.com/twitter/cache-trace.

[2] [n.d.]. Apache Aurora. Retrieved from http://aurora.apache.org/.

[3] [n.d.]. Apache Mesos. Retrieved from http://mesos.apache.org/.

[4] [n.d.]. Apache Traffic Server. Retrieved from https://trafficserver.apache.org/.

[5] [n.d.]. Art. 17 GDPR Right to Erasure ("Right to be Forgotten"). Retrieved from https://gdpr-info.eu/art-17-gdpr/.

[6] [n.d.]. Caching with Twemcache. Retrieved from https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html.

[7] [n.d.]. Database Caching Strategy Using Redis. Retrieved from https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf.

[8] [n.d.]. Decomposing Twitter: Adventures in Service-Oriented Architecture. Retrieved from https://www.infoq.com/presentations/twitter-soa/.

[9] [n.d.]. Do Not Join Lru and Slab Maintainer Threads if They Do Not Exist. Retrieved from https://github.com/memcached/memcached/pull/686.

[10] [n.d.]. Enhance Slab Reallocation for Burst of Evictions. Retrieved from https://github.com/memcached/memcached/pull/695.

[11] [n.d.]. Experiencing Slab OOMs After One Week of Uptime. Retrieved from https://github.com/memcached/memcached/issues/689.

[12] [n.d.]. How To Interpret R-squared and Goodness-of-Fit in Regression Analysis. Retrieved from https://www.datasciencecentral.com/profiles/blogs/regression-analysis-how-do-i-interpret-r-squared-and-assess-the.

[13] [n.d.]. Jemalloc. http://jemalloc.net/.

[14] [n.d.]. Logging Control In W3C Httpd. Retrieved from https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format.

[15] [n.d.]. Memcached - a Distributed Memory Object Caching System. Retrieved from http://memcached.org/.

[16] [n.d.]. Paper Review: MemC3. Retrieved from https://memcached.org/blog/paper-review-memc3/.

[17] [n.d.]. Pelikan. Retrieved from https://github.com/twitter/pelikan.

[18] [n.d.]. Redis. Retrieved from http://redis.io/.

[19] [n.d.]. RocksDB. Retrieved from https://rocksdb.org/.

[20] [n.d.]. Slab Auto-mover Anti-favours Slab 2. Retrieved from https://github.com/memcached/memcached/issues/677.

[21] [n.d.]. Varnish Cache. Retrieved from https://varnish-cache.org/.

[22] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2021. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with Orthus. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association. Retrieved from https://www.usenix.org/conference/fast21/presentation/wu-kan.

[23] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. 2016. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Savannah, GA, 739–753. Retrieved from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/adya.

[24] Mehmet Altinel, Christof Bornhoevd, Chandrasekaran Mohan, Mir Hamid Pirahesh, Berthold Reinwald, and Saileshwar Krishnamurthy. 2008. System and Method for Adaptive Database Caching. U.S. Patent 7,395,258.

[25] Martin Arlitt, Rich Friedrich, and Tai Jin. 1999. Workload characterization of a Web proxy in a cable modem environment. *ACM SIGMETRICS Perform. Eval. Rev.* 27, 2 (1999), 25–36.

[26] Martin Arlitt and Tai Jin. 2000. A workload characterization study of the 1998 world cup web site. *IEEE Netw.* 14, 3 (2000), 30–37.

[27] Martin F. Arlitt and Carey L. Williamson. 1997. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Trans. Netw.* 5, 5 (1997), 631–645.

[28] 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.* 53–64.

[29] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. 1991. Measurements of a distributed file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles.* 198–212.

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

[31] Nathan Beckmann and Daniel Sanchez. 2015. Talus: A simple way to remove cliffs in cache performance. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15).* IEEE, 64–75.

[32] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The CacheLib caching engine: Design and experiences at scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20).* 753–768.

[33] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18).* 195–212.

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

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

[36] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM'99)*, Vol. 1. IEEE, 126–134.

[37] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'13).* 49–60.

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

[39] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. 2020. Characterizing, modeling, and benchmarking RocksDB key-value workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20).* 209–223.

[40] 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 International Conference on Management of Data (SIGMOD'18).* Association for Computing Machinery, New York, NY, 275–290. https://doi.org/10.1145/3183713.3196898

[41] 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 *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20).* USENIX Association, 239–252. Retrieved from https://www.usenix.org/conference/fast20/presentation/chen-jiqiang.

[42] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. 2016. Erasing Belady's limitations: In search of flash cache offline optimality. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC 16).* USENIX Association, 379–392. Retrieved from https://www.usenix.org/conference/atc16/technical-sessions/presentation/cheng.

[43] Yue Cheng, Fred Douglis, Philip Shilane, Grant Wallace, Peter Desnoyers, and Kai Li. 2016. Erasing belady's limitations: In search of flash cache offline optimality. In *Proceedings of the USENIX Annual Technical Conference (USENIX-ATC'16).* 379–392.

[44] Yue Cheng, Aayush Gupta, and Ali R. Butt. 2015. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15).* Association for Computing Machinery, New York, NY, Article 4, 16 pages. https://doi.org/10.1145/2741948.2741967

[45] Ludmila Cherkasova. 1998. *Improving WWW Proxies Performance with Greedy-dual-size-frequency Caching Policy.* Hewlett-Packard Laboratories.

[46] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic cloud caching. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'15)*.

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

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

[49] 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*. 143–154.

[50] Jeffrey Dean and Luiz Andre Barroso. 2013. The tail at scale. *Commun. ACM* 56 (2013), 74–80. Retrieved from http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[51] 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 Symposium on Networked Systems Design and Implementation (NSDI'19)*. 79–94.

[52] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC'19)*. Retrieved from https://www.flux.utah.edu/paper/duplyakin-atc19.

[53] Gil Einziger, Roy Friedman, and Ben Manes. 2017. Tinylfu: A highly efficient cache admission policy. *ACM Trans. Stor.* 13, 4 (2017), 1–31.

[54] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*. 523–535.

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

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

[57] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. 2018. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the 13th EuroSys Conference*. 1–13.

[58] Tyler Estro, Pranav Bhandari, Avani Wildani, and Erez Zadok. 2020. Desperately seeking ... optimal multi-tier cache configurations. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*. USENIX Association. Retrieved from https://www.usenix.org/conference/hotstorage20/presentation/estro.

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

[60] 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*. 1–12.

[61] Wolfram Gloger. [n.d.]. Ptmalloc. Retrieved from http://www.malloc.de/en/.

[62] David A. Holland, Elaine Angelino, Gideon Wald, and Margo I. Seltzer. 2013. Flash caching on the storage client. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'13)*. 127–138.

[63] Yu-Ju Hong and Mithuna Thottethodi. 2013. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC'13)*. Association for Computing Machinery, New York, NY, Article 13, 17 pages. https://doi.org/10.1145/2523616.2525970

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

[65] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 167–181.

[66] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A. Freedman, Ken Birman, and Robbert van Renesse. 2014. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*. 1–7.

[67] Jinho Hwang and Timothy Wood. 2013. Adaptive performance-aware distributed memory caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC'13)*. 33–43.

[68] Sunghwan Ihm and Vivek S. Pai. 2011. Towards understanding modern web traffic. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement Conference*. 295–312.

[69] Yichen Jia, Zili Shao, and Feng Chen. 2020. SlimCache: An efficient data compression scheme for flash-based key-value caching. *ACM Trans. Stor.* 16, 2, Article 14 (June 2020), 34 pages. https://doi.org/10.1145/3383124

[70] 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*. 121–136.

[71] Jaeyeon Jung, Emil Sit, Hari Balakrishnan, and Robert Morris. 2001. DNS performance and the effectiveness of caching. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*. 153–167.

[72] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. 2016. SLIK : Scalable low-latency indexes for a key-value store. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'16)*. 57–70.

[73] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2001. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Comput. Architect. Lett.* 50, 12 (2001), 1352–1361.

[74] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 137–152.

[75] Cheng Li, Philip Shilane, Fred Douglis, Hyong Shim, Stephen Smaldone, and Grant Wallace. 2014. Nitro: A capacity-optimized SSD cache for primary storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'14)*. 501–512.

[76] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. 2015. Pannier: A container-based flash cache for compound objects. In *Proceedings of the 16th Annual Middleware Conference (Middleware'15)*. Association for Computing Machinery, New York, NY, 50–62. https://doi.org/10.1145/2814576.2814734

[77] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. 2017. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Trans. Stor.* 13, 3 (2017), 1–34.

[78] Conglong Li and Alan L Cox. 2015. GD-Wheel: A cost-aware replacement policy for key-value stores. In *Proceedings of the 10th European Conference on Computer Systems*. 1–15.

[79] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. 2015. 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*. 476–488.

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

[81] 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 (FAST'19)*. 143–157.

[82] Qiong Luo, Sailesh Krishnamurthy, C. Mohan, Hamid Pirahesh, Honguk Woo, Bruce G. Lindsay, and Jeffrey F. Naughton. 2002. Middle-tier database caching for e-business. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 600–611.

[83] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'03)*, Vol. 3. 115–130.

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

[85] Elizabeth J. O'neil, Patrick E. O'neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Rec.* 22, 2 (1993), 297–306.

[86] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. 1985. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proceedings of the 10th ACM Symposium on Operating Systems Principles*. 15–24.

[87] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, and Kannan Ramchandran. 2016. EC-Cache: Load-balanced, low-latency cluster caching with online erasure coding. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, Savannah, GA, 401–417. Retrieved from https://www.usenix.org/conference/osdi16/technical-sessions/presentation/rashmi.

[88] Benjamin Reed and Darrell D. E. Long. 1996. Analysis of caching algorithms for distributed file systems. *ACM SIGOPS Operat. Syst. Rev.* 30, 3 (1996), 12–21.

[89] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. 2012. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC'12)*. San Jose, CA. Retrieved from http://www.pdl.cmu.edu/PDL-FTP/CloudComputing/googletrace-socc2012.pdf.

[90] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google Cluster-usage Traces: Format + schema*. Technical Report. Google Inc., Mountain View, CA. Retrieved from https://github.com/google/cluster-data.

[91] John T. Robinson and Murthy V. Devarakonda. 1990. Data cache management using frequency-based replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*. 134–142.

[92] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. USENIX Association, Santa Clara, CA, 1–16. Retrieved from https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble.

[93] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 1–16.

[94] Mohit Saxena, Michael M. Swift, and Yiying Zhang. 2012. Flashtier: A lightweight, consistent and durable storage cache. In *Proceedings of the 7th ACM European Conference on Computer Systems*. 267–280.

[95] Zhaoyan Shen, Feng Chen, Yichen Jia, and Zili Shao. 2018. Didacache: An integration of device and application for flash-based key-value caching. *ACM Trans. Stor.* 14, 3 (2018), 1–32.

[96] Weisong Shi, Randy Wright, Eli Collins, and Vijay Karamcheti. 2002. Workload characterization of a personalized web site and its implications for dynamic content caching. In *Proceedings of the 7th International Workshop on Web Caching and Content Distribution (WCW'02)*. Citeseer.

[97] Ankit Singla, Balakrishnan Chandrasekaran, P. Brighten Godfrey, and Bruce Maggs. 2014. The Internet at the speed of light. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets'14)*. Association for Computing Machinery, New York, NY, 1–7. https://doi.org/10.1145/2670518.2673876

[98] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ : Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 373–386.

[99] Giuseppe Vietri, Liana V. Rodriguez, Wendy A. Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. 2018. Driving cache replacement with ml-based lecar. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'18)*.

[100] Werner Vogels. 1999. File system usage in Windows NT 4.0. *ACM SIGOPS Operat. Syst. Rev.* 33, 5 (1999), 93–109.

[101] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache modeling and optimization using miniature simulations. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'17)*. 487–498.

[102] Qiuping Wang, Jinhong Li, Wen Xia, Erik Kruus, Biplob Debnath, and Patrick P. C. Lee. 2020. Austere flash caching with deduplication and compression. In *Proceedings of the USENIX Annual Technical Conference (USENIXATC'20)*. 713–726.

[103] Terry A. Welch. 1984. A technique for high-performance data compression. *Computer* 6 (1984), 8–19.

[104] Patrick Wendell and Michael J. Freedman. 2011. Going viral: Flash crowds in an open CDN. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement Conference*. 549–558.

[105] Wikimedia. [n.d.]. Analytics/Data Lake/Traffic/Caching. Retrieved from https://wikitech.wikimedia.org/wiki/Analytics/Data_Lake/Traffic/Caching.

[106] Wikimedia. [n.d.]. Caching Overview—Wikitech. Retrieved from https://wikitech.wikimedia.org/wiki/Caching_overview.

[107] John Wilkes. 2011. More Google Cluster Data. Google research blog. Retrieved from http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html.

[108] Juncheng Yang. [n.d.]. libCacheSim. Retrieved from https://github.com/1a1a11a/libCacheSim.

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

[110] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large-scale analysis of hundreds of in-memory cache clusters at Twitter. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, 191–208. Retrieved from https://www.usenix.org/conference/osdi20/presentation/yang.

[111] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2021. Segcache: Memory-efficient and high-throughput DRAM cache for small objects. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI'21)*. USENIX Association. Retrieved from https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng.

[112] Lei Zhang, Reza Karimi, Irfan Ahmad, and Ymir Vigfusson. 2020. Optimal data placement for heterogeneous cache, memory, and storage systems. *Proc. ACM Measure. Anal. Comput. Syst.* 4, 1 (2020), 1–27.

[113] Lei Zhang, Juncheng Yang, Anna Blasiak, Mike McCall, and Ymir Vigfusson. 2020. When is the cache warm? Manufacturing a rule of thumb. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20).*

[114] Wei Zhang, Jinho Hwang, Timothy Wood, K. K. Ramakrishnan, and Howie Huang. 2014. Load balancing of heterogeneous workloads in memcached clusters. In *Proceedings of the 9th International Workshop on Feedback Computing (Feedback Computing 14).*

[115] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. 2018. Demystifying cache policies for photo stores at scale: A tencent case study. In *Proceedings of the International Conference on Supercomputing*. 284–294.

[116] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference, General Track*. 91–104.

[117] Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter, and Michael A. Kozuch. 2012. Saving cash by using less cache. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12).*

Received February 2021; accepted May 2021