Characterizing Facebook's Memcached Workload

Yuehai Xu (Wayne State University, Facebook) yhxu@wayne.edu
Eitan Frachtenberg (Facebook) etc@fb.com
Song Jiang (Wayne State University) sjiang@wayne.edu
Mike Paleczny (Facebook) mpal@fb.com

Abstract

This article analyzes the workload of Memcached at Facebook, one of the world's largest key-value deployments. We look at server-side performance, request composition, caching efficacy, and key locality. Our observations lead to several design insights and new research directions for key value caches, such as the relative inadequacy of the least-recently-used replacement policy.

1 Introduction

Contemporary Web sites can store and process very large amounts of data. To provide timely service to their users, many internet products have adopted a simple but effective caching infrastructure atop the conventional databases that store these data. Called key-value (KV) stores, these caches store and supply information that is cheaper or faster to cache than to re-obtain, such as commonly accessed results of database queries or the results of complex computations that require temporary storage and distribution. In a KV-cache system, data are organized in ordered (key, value) pairs and usually stored in a number of servers, essentially forming a distributed hash table. Various KV-cache implementations have been developed and deployed in large-scale Internet services, including Dynamo at Amazon [7]; Redis at GitHub, Digg, and Blizzard Interactive [1]; Memcached at Facebook, Zynga and Twitter [5, 14]; and Voldemort at Linkedin [2].

Because many data requests exhibit some form of locality, allowing a popular subset of data to be identified and predicted, a substantial amount of database operations can be replaced by quick in-memory lookups, for significantly reduced response time. To provide this performance boost, KV caches are carefully tuned to minimize response times and maximize the probability of caching request data

(or hit rate). But like all caching heuristics, a KV-cache's performance is highly dependent on its workload. It is therefore essential to understand the workload's characteristics in order to understand and improve the cache's design.

In addition, analyzing such workloads can: offer insights into the role and effectiveness of memory-based caching in distributed website infrastructure; expose the underlying patterns of user behavior; and provide difficult-to-obtain data and statistical distributions for future studies. But many such workloads are proprietary and hard to access, especially those of very large scale. Such analyses are therefore rare and the workload characteristics are usually assumed in academic research and system design without substantial support from empirical evidence.

In this article we discuss five such workloads from Facebook's Memcached deployment. Aside from the sheer scale of the site and data (over 284 billion requests over a period of 58 sample days), this case study also provides a description of several different usage scenarios for KV caches. This variability serves to explore the relationship between the cache and various data domains: where overall site patterns are adequately handled by a generalized caching infrastructure, and where specialization would help. But first, let us start by describing how a KV cache such as Memcached is used in practice.

2 Anatomy of a Large-Scale Social Network

Many Web services such as social networks, email, maps, and retailers must store large amounts of data and retrieve specific items on demand very quickly. Facebook, for example, stores basic profile information for each of its users, as well as content they post, individual privacy settings, etc. When a user logs in to Facebook's main page and is presented with a newsfeed of their connections and interests, hundreds or thousands of such data items must be retrieved, aggregated, filtered, ranked, and presented in a very short time. The total amount of potential data to retrieve for all users is so large that it is impractical to store an entire copy locally on each web server that takes user requests. Instead, we must rely on a distributed storage scheme, wherein multiple storage servers are shared among all Web servers

The persistent storage itself takes place in the form of multiple shards and copies of a relational database called MySQL. MySQL has been carefully tuned to maximize throughput and lower latency for high loads, but its performance can be limited by the underlying storage layer, typically hard drives or flash. The solution is caching—the selective and temporary storage of a subset of data on faster RAM. Caching works when some items are much more likely to be requested than others. By provisioning enough RAM to cache the desired amount of popular items, we can create a customizable blend of performance and resource tradeoffs.

Tab. 1: Memcached pools sampled (in one cluster), including their typical deployment sizes, read request rates, and average hit rates. The pool names do not match their UNIX namesakes, but are used for illustrative purposes here instead of their internal names.

Pool	**Size**	**GET/s**	Hit Rate	**Description**
USR	few	100,500	98.2%	user-account status information
APP	dozens	65,800	92.9%	object metadata of one application
ETC	hundreds	57,800	81.4%	nonspecific, general-purpose
VAR	dozens	73,700	93.7%	server-side browser information
SYS	few	7,200	98.7%	system data on service location

2.1 Software Architecture

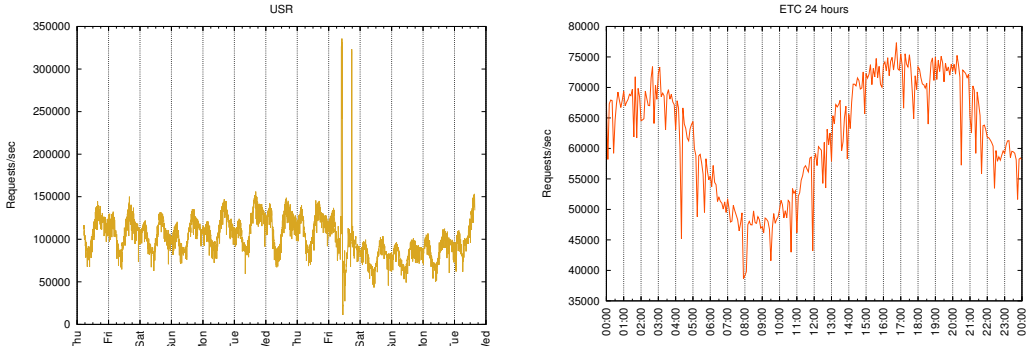
Memcached[1] is an open-source software package that exposes data in RAM to clients over the network. As data size grows in the application, more RAM can be added to a server, or more servers can be added to the network. Additional servers generally only communicate with clients. Clients use consistent hashing [6] to select a unique server per *key*, requiring only the knowledge of the total number of servers and their IP addresses. This technique presents the entire aggregate data in the servers as a unified distributed hash table, keeps servers completely independent, and facilitates scaling as data size grows.

Memcached's interface provides the basic primitives that hash tables provide, as well as more complex operations built atop them. The two basic operations are *GET*, to fetch the value of a given key, and *SET* to cache a value for a given key (typically after a previous GET failure, since Memcached is used as a look-aside, demand-filled cache). Another common operation for data backed by persistent storage is to *DELETE* a key-value pair as a way to invalidate the key if it was modified in persistent storage. To make room for new items after the cache fills up, older items are evicted using the least-recently-used (LRU) algorithm [6].

2.2 Deployment

Physically, Facebook deploys front-end servers in multiple datacenters, each containing one or more *clusters* of varying sizes. Front-end clusters consist of both Web servers and caching servers, including Memcached. These servers are further subdivided based on the concept of *pools*. A pool defines a class of Memcached keys. Pools are used to separate the total possible key space into buckets, allowing better efficiency by grouping keys of a single application, with similar access patterns and data requirements. Any given key will be uniquely mapped to a single pool by the key's prefix, which identifies an application.

[1] `http://memcached.org/`



(a) (b)

Fig. 1: (a) Request rates at different days (USR) and (b) times of day (ETC, Coordinated Universal Time—UTC). Each data point counts the total number of requests in the preceding second.

We analyzed one trace each from five separate pools. These pools represent a varied spectrum of application domains and cache usage characteristics (Table 1). We traced all Memcached packets on these servers using a custom kernel module [5] and collected between $3TB$ to $7TB$ of trace data from each server, representing at least a week's worth of consecutive samples. All five Memcached instances ran on identical hardware.

3 Request Rates and Composition

3.1 Request Rates

Many companies rely on Memcached to serve terabytes of data in aggregate every day, over many millions of requests. Average sustained request rates can reach $100,000+$ requests per second, as shown in Table 1. These request rates represent relatively modest network bandwidth. But Memcached's performance capacity must accommodate significantly more headroom than mean sustained rates. Fig. 1(a) shows that in extreme cases for USR, the transient request rate can more than triple the sustained rate. These outliers stem from a variety of sources, including high transient interest in specific events, highly popular keys on individual servers, and operational issues. Consequently, when analyzing Memcached's performance, we focus on sustained end-to-end latency and maximum sustained request rate (while meeting latency constraints), and not on network bandwidth [6].

Figure. 1 also reveals how Memcached's load varies normally over time. USR's 13-day trace shows a recurring daily pattern, as well as a weekly pattern

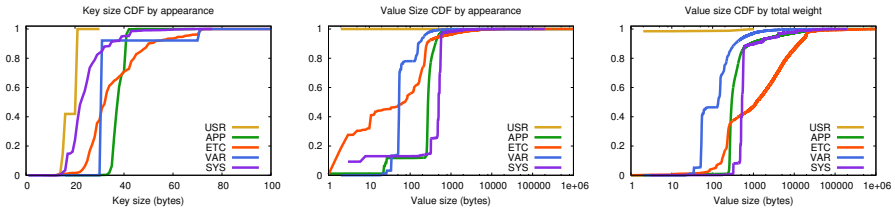


Fig. 2: Key and value size distributions for all traces. The leftmost cumulative distribution function (CDF) shows the sizes of keys, up to Memcached's limit of 250 B (not shown). The center plot similarly shows how value sizes distribute. The rightmost CDF aggregates value sizes by the total amount of data they use in the cache, so for example, values under 320 B or so in SM use virtually no space in the cache; 320 B values weigh around 8% of the data, and values close to 500 B take up nearly 80% of the entire cache's allocation for values.

that exhibits a somewhat lower load approaching the weekend. All other traces exhibit similar daily patterns, but with different values and amplitudes. If we zoom in on one day for ETC for example (righthand figure), we notice that request rates bottom out around 08:00 UTC and have two peaks around 17:00 and 03:00. Although different traces (and sometimes even different days in the same trace) differ in which of the two peaks is higher, the entire period between them, representing the Western Hemisphere daytime, exhibits the highest traffic volume.

3.2 Request Sizes

Next, we turn our attention to the sizes of keys and values in each pool (Fig. 2) for SET requests (GET requests have identical sizes for hits, and zero data size for misses). All distributions show strong modalities. For example, over 90% of APP's keys are 31 bytes long, and values sizes around 270 B show up in more than 30% of SET requests. USR is the most extreme: it only has two key size values (16 B and 21 B) and virtually just one value size (2 B). Even in ETC, the most heterogeneous of the pools, requests with 2-, 3-, or 11-byte values add up to 40% of the total requests. On the other hand, it also has a few very large values (around 1MB) that skew the weight distribution (rightmost plot in Fig. 2), leaving less caching space for smaller values. Small values dominate all workloads, not just in count, but especially in overall weight. Except for ETC, 90% of all Memcached's data space is allocated to values of less than 500 B.

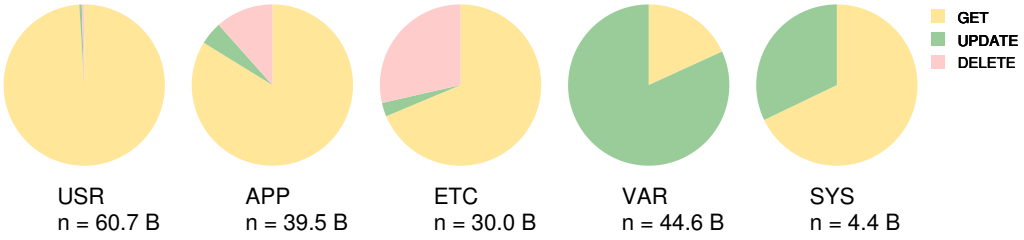


Fig. 3: Distribution of request types per pool, over exactly 7 days. UPDATE commands aggregate all non-DELETE writing operations.

3.3 Request Composition

Last, we look at the composition of basic request types that comprise the workload (Fig. 3) and make the following observations:

USR handles significantly more GET requests than any of the other pools (at an approximately 30 : 1 ratio). GET operations comprise over 99.8% of this pool's workload. One reason for this is that the pool is sized large enough to maximize hit rates, so refreshing values is rarely necessary. These values are also updated at a slower rate than some of the other pools. The overall effect is that USR is used more like RAM-based persistent storage than a cache.

APP has high absolute and relative GET rates too—owing to the popularity of this application. But also has a large number of DELETE operations, which occur when a cached database entry is modified (but not required to be set again in the cache). SET operations occur when the Web servers add a value to the cache. The relatively high number of DELETE operations show that this pool represents database-backed values that are affected by frequent user modifications.

ETC has similar characteristics to APP, but with a higher fraction of DELETE requests (of which not all are currently cached, and therefore miss). ETC is the largest and least specific of the pools, so its workloads might be the most representative to emulate. Because of its general applicability to mixed workloads, it has been modeled in detail in [5].

VAR is the only pool of the five that is write-dominated. It stores short-term values such as browser-window size for opportunistic latency reduction. As such, these values are not backed by a database (hence, no invalidating DELETEs are required). But they change frequently, accounting for the high number of UPDATEs.

SYS is used to locate servers and services, not user data. As such, the number of requests scales with the number of servers, not the number of user

requests, which is much larger. This explains why the total number of SYS requests is much smaller than the other pools'.

3.4 Discussion

We found that Memcached requests exhibit clear modality in request sizes, with a strong bias for small values. We also observed temporal patterns in request rates that are mostly predictable and require low bandwidth, but can still experience very significant outliers of transient high load. There are several implications for cache design and system optimizations from these data:

1. Network overhead in the processing of multiple small packets can be substantial relative to payload, which explains why Facebook coalesces as many requests as possible in as few packets as possible [6].

2. Memcached allocates memory for KV values in slabs of fixed size units. The strong modality of each workload implies that different Memcached pools can optimize memory allocation by modifying the slab size constants to fit each distribution. In practice, this is an unmanageable and unscalable solution, so instead Memcached uses 44 different slab classes with exponentially growing sizes to reduce allocation waste, especially for small sizes. This does, however, result in some memory fragmentation.

3. The ratio of GETs to UPDATEs in ETC can be very high—significantly higher in fact than most synthetic workloads typically assume. For demand-filled caches where each miss is followed by an UPDATE, the ratios of GET to UPDATE operations mentioned above are related to hit rate in general and the relative size of the cache to the data in particular. So in theory, one could justify any synthetic GET to UPDATE mix by controlling the cache size. But in practice, not all caches or keys are demand-filled, and these caches are already sized to fit a real-world workload in a way that successfully trades off hit rates to cost.

These observations on the nature of the cache lead naturally to the next question (and section): how effective is Memcached at servicing its GET workload—its *raison d'être.*

4 Cache Effectiveness

Understanding cache effectiveness can be broken down to the following questions: how well does Memcached service GET requests for the various workloads? what factors affect good cache performance? what characterizes poor cache performance, and what can we do to improve it?

The main metric used in evaluating cache efficacy is hit rate: the percentage of GET requests that return a value. Hit rate is determined by three factors:

available storage (which is fixed, in our discussion); the patterns of the underlying workload and their predictability; and how well the cache policies utilize the available space and match these patterns to store items with a high probability of recall. Understanding the sources of misses will then offer insights into why and when the cache wasn't able to predict a future item. We then look deeper into the workload's statistical properties to understand how amenable it is to this prediction in the first place. The overall hit rate of each server, as derived from the traces and verified with Memcached's own statistics, are shown in Table 1.

SYS and USR exhibit very high hit rates. Recall from that same table that these are also the smallest pools, so the entire keyspace can be stored with relatively few resources, thus eliminating all space constraints from hit rates. Next down in hit-rate ranking are APP and VAR, which are larger pools, and finally, ETC, the largest pool, also exhibits the lowest hit rate. So can pool size completely explain hit rates? Is there anything we could do to increase hit rates except buy more memory? To answer these questions, we take a deeper dive into workload patterns and composition.

4.1 Sources of Misses

To understand hit rate, it is instructive to analyze its complement, miss rate, and specifically to try to understand the sources for cache misses. These sources can tell us if there are any latent hits that can still be exploited, and possibly even how.

We distinguish three types of misses:

- *Compulsory misses* are caused by keys that have never been requested before (or at least not in a very long time). In a demand-filled cache with no prefetching like Memcached, no keys populate the cache until they have been requested at least once, so as long as the workload introduce new keys, there is not much we can do about these misses.
- *Invalidation misses* occur when a requested value had been in the cache before, but was removed by a subsequent DELETE request.
- *Eviction (capacity) misses* represent keys that had been in the cache, but were evicted by the replacement policy before the next access. If most misses are of this kind, then indeed the combination of pool size and storage size can explain hit rates.

Several interesting observations can be made. The first is that VAR and SYS have virtually 100% compulsory misses. Invalidation misses are absent because these pools are not database-backed, and eviction misses are nearly non-existent because of ample space provisioning. Therefore, keys are invariably missed only upon the first request, or when new keys are added.

On the opposite end, about 87% of USR's misses are caused by evictions. This is puzzling, since USR is the smallest of pools, enabling sufficient RAM

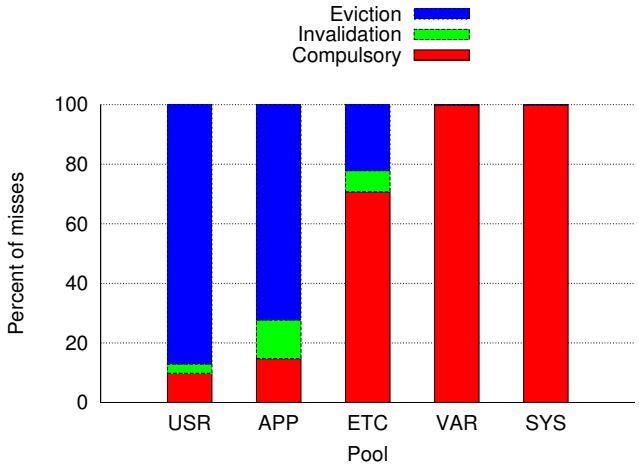


Fig. 4: Distribution of cache miss causes per pool.

provisioning to cover the entire key space. This larger percentage of eviction misses originates from service jobs that request sections of the key space with little discernible locality, such as data validation or migration to a new format. So the cache replacement policy has little effect in predicting future key accesses to these keys and preventing eviction misses.

At last we come to ETC and APP, the two largest pools, with 22% and 72% eviction misses, respectively. One straightforward measure to improve hit rates in these two pools would be to increase the total amount of memory in their server pool, permitting fewer keys to be evicted. But this solution obviously costs more money and will help little if the replacement policy continues to accumulate rarely used keys. A better solution would be to improve the replacement policy to keep valuable items longer, and quickly evict items that are less likely to be recalled soon. To understand whether alternative replacement policies would better serve the workload patterns, we next examine these patterns in terms of their key reuse over time, also known as temporal locality.

4.2 Temporal Locality Measures

We start by looking at how skewed is the key popularity distribution, measured as a ratio of each key's GET requests from the total (Fig. 5). All workloads exhibit long-tailed popularity distributions. For example, 50% of ETC's keys (and 40% of APP's) occur in no more than 1% of all requests, meaning they do not repeat many times, while a few popular keys repeat in millions of requests per day. This high concentration of repeating keys is what makes caching economical in the first place. SYS is the exception to the rule, as its values are cached locally by clients, which could explain why some 65% of its keys hardly

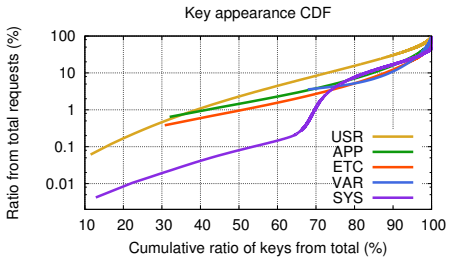


Fig. 5: CDF of key appearances, depicting how many keys account for how many requests, in relative terms. Keys are sorted from least popular to most popular.

repeat at all.

We can conceivably use these skewed distributions to improve the replacement policy: By evicting unpopular keys sooner, instead of letting them linger in memory until expired by LRU, a full cycle of insertions later, we could leave room for more popular keys, thus increasing hit rate. For example, about a fifth of all of APP's and ETC's keys are only requested at most once in any given hour. The challenge is telling the two classes of keys apart, when we don't have a-priori knowledge of their popularity.

One clue to key popularity can be measured in reuse period—the time between consecutive accesses to the key. Fig. 6 counts all key accesses and bins them according to the time duration from the previous access to each key. Unique keys (those that do not repeat at all within the trace period) are excluded from this count. The figure shows that key repeatability is highly localized and bursty, with some daily patterns (likely corresponding to some users always logging in at the same time of day). For the ETC trace, for example, 88.5% of the keys are reused within an hour, but only 4% more within two, and within six hours 96.4% of all non-unique keys have already repeated. The main takeaway from this chart is that reuse period decays at an exponential rate. This implies diminishing returns to a strategy of increasing memory resources beyond a certain point, because if we can already cache most keys appearing in a given time windows, and double it with twice the memory capacity, only a shrinking fraction of the keys that would have otherwise been evicted would repeat again in the new, larger time window.

As before, the SYS pool stands out. It doesn't show the same 24-hour periodicity as the other pools, because its keys relate to servers and services, not users. It also decays faster than the others. Again, since its data are cached locally by clients, it is likely that most of SYS's GET requests represent data that are newly available, updated, or expired from the client cache; these are then requested by many clients concurrently. This would explain why 99.9% of GET requests are repeated within an hour of the first key access. Later, such

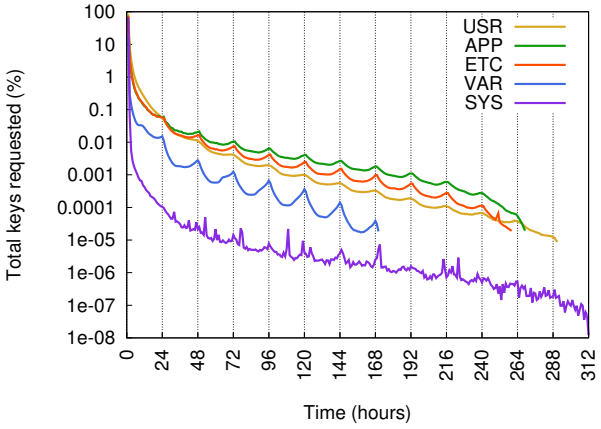


Fig. 6: Reuse period histogram per pool. Each hour-long bin n counts keys that were first requested n hours after their latest appearance. Keys can add counts to multiple bins if they occur more than twice.

keys would be cached locally and accessed rarely, perhaps when a newly added client needs to fill its own cache.

5 Related Work

A core element of any caching system is its replacement algorithm. By analyzing the workloads' locality, source of misses, and request sizes, our original paper suggested areas where an optimized replacement strategy could help. In fact, some of these optimizations have since been reportedly implemented [14], including an adaptive allocator to periodically rebalance the slab allocator, and the use of expiration time associated data items for early eviction of some short-lived keys.

Caching as a general research topic has been extensively studied. The LRU algorithm, which is adopted in Memcached, has been shown to have several weaknesses, and a number of algorithms have been proposed to improve it. The 2Q algorithm was proposed to evict cold data earlier from the cache so that relatively warm data can stay longer [10]. The LRFU algorithm introduced access frequency into the LRU algorithm, to improve its replacement decisions on data with distinct access frequencies [11]. These weaknesses of LRU also show in this workload study. To address both weaknesses with an efficient implementation, Jiang and Zhang proposed the LIRS algorithm to explicitly use reuse distance—in principle equivalent to the reuse period measured in this paper—to quantify locality and choose eviction victims [9]. The LRU algorithm requires a lock to maintain the integrity of its data structure, which can lead to a performance bottleneck in a highly contended environment such as Memcached's. In

contrast, the CLOCK algorithm eliminates this need while maintaining similar performance to that of LRU. The CLOCK-Pro algorithm, which also removes this lock, has a performance as good as that LIRS' [8].

Web caches are another area of active research. In a study of requests received by Web servers, Arlitt and Williamson found that 80% of requested documents are smaller than $\approx 10KB$. However, requests to these documents generate only 26% of data bytes retrieved from the server [4]. This finding is consistent with the distribution we describe in [5].

KV caches are also receiving ample attention in the literature, covering aspects such as performance, energy efficiency, and cost effectiveness [6, 15]. Absent well-publicized workload traces, in particular large-scale production traces, many works used hypothetical or synthetic workloads [15]. For example, to evaluate SILT, a KV-cache design that constructs a three-level store hierarchy for storage on flash memory with a memory based index, the authors assumed a workload of 10% SET and 90% GET requests using $20B$ keys and $100B$ values, as well as a workload of 50% SET and 50% GET requests for $64B$ KV pairs [13]. In the evaluation of CLAM, a KV-cache design that places both hash table and data items on flash, the authors used synthetic workloads that generate keys from a random distribution and a number of artificial workload mixes [3]. Additional references appear in the conference version of this paper [5]. That paper also contains a more detailed statistical description of ETC, which has later been used to construct a synthetic workload generator by a Stanford team [12].

6 Summary and Future Work

This paper exposes five workloads from one of the world's largest KV-cache deployments. These five workloads exhibit both common and idiosyncratic properties that must factor into the design of effective large-scale caching systems. For example, all user-related caches exhibit diurnal and weekly cycles that correspond to users' content consumption; but the amplitude and presence of outliers can vary dramatically from one workload to the next. We also investigate at depth the properties that make some workloads easier or harder to cache effectively with Memcached. For example, all workloads but one (SYS) exhibit very strong temporal and "spatial" locality. But each workload has a different composition of requests (particularly the missing ones) that determine and bound the cap for potential hit-rate improvements.

One particular workload, ETC, is interesting and useful to analyze because it is the closest workload to a general-purpose one, i.e., not tied to any one specific Facebook application. The description of its performance and locality characteristics can therefore serve other researchers in constructing more realistic KV-cache models and synthetic workloads.

Looking at hit rates has shown that there is room for improvement, especially with the largest pools, ETC and APP. By analyzing the types and distribution of misses, we were able to quantify precisely the potential for additional hits.

They are the fraction of GET request that miss because of lack of capacity: 4.1% in ETC (22% of the 18.4% miss rate) and 5.1% in APP (72% of 7.1% misses). This potential may sound modest, but it represents over 120 million GET requests per day per server, with noticeable impact on service latency.

There are two possible approaches to tackle capacity misses: increasing capacity or improving the logic that controls the composition of the cache. The former is expensive and yields diminishing returns (Fig. 6). Recall that within 6 hours, 96% of GET requests in ETC that would be repeated at all, have already repeated—far above the 81.2% hit rate. On the other hand, non-repeating keys—or those who grow cold and stop repeating—still occur in abundance and take up significant cache space. LRU is not an ideal replacement policy for these keys, which has been demonstrated in all five pools. And since all pools exhibit strong temporal locality, even those pools with adequate memory capacity could benefit from better eviction prediction, for example by reducing the amount of memory (and cost) required by these machines.

One of our directions of future investigation is therefore to replace Memcached's replacement policy. One approach could be to assume that keys that don't repeat within a short time period are likely cold keys, and evict them sooner. Another open question is whether the bursty access pattern of most repeating keys can be exploited to identify when keys grow cold, even if initially requested many times, and evict them sooner.

Acknowledgements

This work is based on an earlier work: "Workload Analysis of a Large-Scale Key-Value Store", in SIGMETRICS/Perfomance'12 © ACM, 2012. `http://doi.acm.org/10.1145/2254756.2254766`

References

[1] `http://redis.io`.

[2] `voldemort-project.com`.

[3] Ashok Anand, Chitra Muthukrishnan, Steven Kappes, Aditya Akella, and Suman Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proceedings of the 7th USENIX conference on Networked Systems Design and Implementation*, April 2010.

[4] Martin F. Arlitt and Carey L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5:631–645, October 1997.

[5] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th Joint Conference On Measurement And Modeling of Computer Systems (SIGMETRICS/Performance'12)*, London, UK, June 2012. ACM. `frachtenberg.org/eitan/pubs/`.

[6] Mateusz Berezecki, Eitan Frachtenberg, Mike Paleczny, and Kenneth Steele. Many-core key-value store. In *Proceedings of the Second International Green Computing Conference*, Orlando, FL, August 2011. IEEE. `frachtenberg.org/eitan/pubs/`.

[7] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 205–220, Stevenson, WA, 2007. `citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.127.6956`.

[8] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the CLOCK replacement. In *Proceedings of the USENIX Annual Technical Conference*, pages 323–336, April 2005.

[9] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS'02, pages 31–42. ACM, 2002.

[10] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB'94)*, pages 439–450, September 1994.

[11] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 134–143, June 1999.

[12] Jacob Leverich. Mutilate: high-performance memcached load generator. `github.com/leverich/mutilate`.

[13] Hyeontaek Lim, Bin Fan, David G. Andersen, and Michael Kaminsky. Silt: A memory-eficient, high-performance key-value store. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, October 2011.

[14] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab,

David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In *Tenth USENIX Symposium on Networked Systems Design and Implementation*, Lombard, IL, April 2013.

[15] Vijay R. Vasudevan. *Energy-Efficient Data-intensive Computing with a Fast Array of Wimpy Nodes*. PhD thesis, Carnegie Mellon University, October 2011.