A Penalty Aware Memory Allocation Scheme for Key-value Cache

Jianqiang Ou, Marc Patton, Michael Devon Moore, Yuehai Xu, and Song Jiang
The ECE Department, Wayne State University,
Detroit, MI, 48202
Email: {*jianqiang.ou, marc.patton, michael.moore9, yhxu, sjiang*}@wayne.edu

***Abstract*—Key-value caches, represented by Memcached, play a critical role in data centers. Its efficacy can significantly impact users' perceived service time and back-end systems' workloads. A central issue in the in-memory cache's management is memory allocation, or how the limited space is distributed for storing key-value items of various sizes. When a cache is full, the allocation issue is how to conduct replacement operations on items of different sizes. To effectively address the issue, a practitioner must simultaneously consider three factors, which are access locality, item size, and miss penalty. Existing designs consider only one or two of the first two factors, and pay little attention on miss penalty. This inadequacy can substantially compromise utilization of cache space and request service time.**

In this paper we propose a Penalty Aware Memory Allocation scheme (PAMA) that takes all three factors into account. While the three different factors cannot be directly compared to each other in a quantitative manner, PAMA uses their impacts on service time to determine where a unit of memory space should be (de)allocated. The impacts are quantified as the decrease (or increase) of service time if a unit of space is allocated (or deallocated). PAMA efficiently tracks access pattern and use of memory, and speculatively evaluates the impacts to enable penalty-aware memory allocation for KV caches. Our evaluation with real-world Memcached workload traces demonstrates that PAMA can significantly reduce request service time compared to other representative KV cache management schemes.

***Keywords*—Key-value Cache; Replacement Algorithm; Locality.**

I. Introduction

As a critical component of today's data center infrastructure, key-value (KV) cache amasses a large collection of memory distributed on a cluster of servers and uses it as a cache to accelerate front-end applications. Unlike conventional caches, which are usually used to bridge access speed gaps between fast and expensive memory and slow but more affordable storage devices, the KV cache is used to store data objects, which often represent expensive-to-compute values, such as results of popular database queries. Because it could take a much longer time, such as many milliseconds, to obtain the values, by caching them one can significantly reduce the load on the back-end database systems, accelerate frond-end applications, and improve customers' user experience.

Today KV cache systems have been deployed in almost all major Internet companies. Among the systems, Memcached is one of the most known ones [4] and has been deployed in companies including Facebook, Zynga, and Twitter. These systems are object stores, where objects are in the form of (key, value) pairs. Their interfaces usually provide simple primitives, such as insertion (SET), retrieval (GET), and deletion (DEL). Just like any cache systems, a KV cache aims to achieve as many hits as possible by maximizing the utilization of limited cache space. In the meantime, a KV cache has its unique characteristics that are different from caches traditionally deployed for caching data on the block storage devices, and demands different considerations in its design.

First, size of KV items in the cache can be distributed over a very large range. Our study of Facebook's production workload on its Memcached system has shown that item size can be as small as a couple of bytes and as large as 1MB [6]. Hit ratio, as a generic metric often used for evaluating caching efficacy, measures number of hits over a given number of GET requests. However, it does not consider the cost of producing the hits, where the cost is the space held by the hit items. Therefore, in the design of a replacement algorithm for the cache, items that have been requested many times do not necessarily warrant their long-term stay in the cache without being evicted, especially when they are very large.

Second, temporal access pattern can regularly change. The Memcached workload study shows clear diurnal patterns [6]. The load variation during a diurnal cycle can be on the order of two times. The pattern can also be drastically affected by the occurrence of major news or media events leading to swift surge of access frequency to certain KV items. We believe similar observations are likely to exist in other KV systems supporting Internet-wide services. Such workload behaviors suggest that the cache's replacement algorithm must be responsive to the pattern changes. In particular, when cache space is allocated to regions holding items of different sizes, the space has to be responsively allocated/re-allocated between the regions.

Third, miss penalty can vary in a large range. A cache is employed to avoid misses, or more generally, to minimize miss penalty. Replacement algorithms used in most storage caches aim to only minimize misses (or miss ratio), rather than miss penalty. While this can cause a substantial performance issue for some block devices, such as hard disks [10], the inadequacy can have a much more serious impact on KV cache's performance. Figure 1 shows miss penalties with KV items of different sizes for one of the Facebook workloads we had studied ("APP") [6]. While the workload traces do not explicitly contain miss time of a GET request, we assume that a GET request miss immediately follows a retrieval of the missing value from a back-end system and a SET request for caching the corresponding KV item into the cache. Accordingly, we consider the time period between the GET

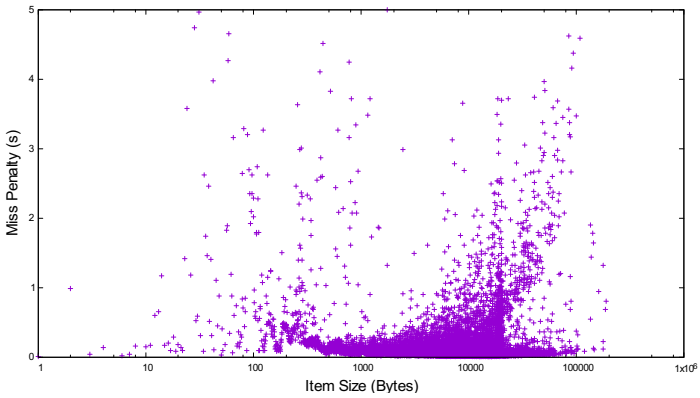

Fig. 1: Miss penalties of GET requests for KV items of different sizes.

miss and the PUT request as the corresponding miss penalty of the GET request. As shown, the penalty for a miss can be as small as a few milliseconds and as large as several seconds. While the metric truly relevant to a KV-cache's users is request service time, excluding the penalty in the design of a cache management system can possibly lead to many (very-)high-penalty misses and much deteriorated cache performance.

In this paper we propose a dynamic memory (re)-allocation scheme that takes all three factors affecting request service time into account, which are locality, size, and miss penalty. While the three factors seem to be hard to be directly integrated into one quantity guiding the memory allocation optimized for minimal request service time, we propose to measure the contribution on the reduction of miss penalty by a unit of cache space, named as slab, in a time window. This is equivalent to measure the contribution on the amount of total miss penalty experienced by users should the KV items held in the slab not be cached in the time window. We name this measure as the *value* of the slab. For a slab of memory to be of high value, the slab can contain frequently accessed items (of strong locality), a large number of items (of small size), and/or items of high miss penalties. Using the measure accommodating miss penalty, we design a Penalty Aware Memory Allocation scheme, named as PAMA. In the scheme, a slab's value is efficiently estimated or predicted to enable the most effective use of cache space to minimize request service times. We use a KV cache simulator to evaluate the PAMA scheme and other representative schemes with traces collected on Facebook's production Memcached systems and observe significant performance improvements by PAMA.

The rest of this paper is organized as follows. We describe background and related work of KV caching in Section II. We then describe design of the PAMA scheme in Section III. Section IV provides a comprehensive evaluation, and Section V concludes.

II. Background and Related Work

In a KV store each data item includes a key, a value, and metadata, is stored individually, and is often indexed with a hash table. As the item size can vary in a large range, cache space is not allocated in a fixed unit. This avoids significant memory fragmentation. To this end, Memcached places items in different queues, called *classes* in the Memcached community, each dedicated to items in a certain size range. Memory space is allocated to different classes in a fixed unit called a *slab*. A slab is divided into one or multiple equally-sized slots, depending on which class it is allocated to, and each slot holds one item in the corresponding class. When a new item is admitted into a class matching its size, Memcached first searches the class for a free slot to accommodate the item. If this search fails, a free slab is requested and adds into the class. If such a slab is not granted, a replacement slot within the class needs to identified and evicted often by using the LRU (Least Recently Used) replacement algorithm. There are two scenarios where such a slab can be found. One is the system has not yet used up all of its available memory space and allocates a free slab to the class. The other is that all memory space has been allocated and a slab is taken from another class by evicting all its holding items. The allocation and reallocation of slabs in a KV store are essential to the cache's efficacy.

In the earlier versions of Memcached, the aforementioned second scenario is not allowed. After the initial memory space is exhausted, the allocations to the classes will not change. Apparently the allocations are determined by initial workload when the system is being warmed up. They remain fixed after this initial phase even when workload characteristics change significantly, leading to serious under-utilization of cache space. To address this issue, there have been a number of designs recently proposed.

In a recent version of Memcached, relocations among classes are allowed [5]. However, the relocation policy is conservative and such relocations do not occur often. In every time window of ten minutes, the number of misses in each class are recorded. If a class continuously receives the largest number of misses for three times, and there exists a class that does not see any misses in the three time windows, a slab is migrated from the class without misses to the class with the most misses. While this design allows memory space to be reallocated, it does not fully consider locality, size, and miss penalty. Regarding locality, if misses on a class are produced by a flush of cold items (that are unlikely to be re-accessed soon), the additional slabs received by the class would not be efficiently used. In contrast, PAMA estimates the number of misses that would be avoided in a class if it gains a new slab, or that could be produced in a class if it loses a slab. Regarding size, if a class for large items generates many misses, possibly due to attempting to access nonexistent items, it would grab a large number of slabs in a short period of time and make classes for smaller items lose their still useful items. Regarding miss penalty, this information is not considered at all, though it can be approximated by the duration between the missing GET and its subsequent SET that reinstates the value.

A team at Facebook improves the original Memcached by re-balancing slab allocations across classes [11]. The optimized Memcached attempts to balance the age of LRU items in different classes to approximate a single global LRU replacement policy for the entire cache. Here an item's age refers to its latest access time. Specifically, if the scheme finds that the age of a class's LRU item is 20% younger than the average age of the other classes' LRU items, a slab is moved from the class with

the oldest LRU item to the class with the youngest LRU item. In this way, locality can be better exploited. However, it still does not consider item size and miss penalty.

Memcached was also optimized at Twitter into a new version called Twemcache [3]. Twemcache has a slab reallocation policy that aggressively moves slabs between classes. The strategy is very simple: when a class has a miss but does not have free space, Twemcache chooses a random class and reassigns one of its slabs to the class with the miss. By doing this, Twemcache tries to evenly spread misses across the classes. However, in the case that all slabs in a class are efficiently used but few misses are produced, the class's slabs should not be taken away.

The periodic slab allocation (PSA) policy tries to equalize request density, or number of requests per slab, across different classes [2]. For every M misses, where M is a predefined constant, PSA relocates a slab from the class with the lowest density to the one with the largest number of misses recorded in a time window. By normalizing number of requests (or misses) over space size, PSA takes item size into its consideration, though it still ignores the impact of miss penalty on the cache performance. In addition, in a class, different slabs can have different density, and the LRU slab, which is the candidate for relocation, may not be the one with the lowest density. Therefore, using the density to exploit locality is likely to compromise cache performance.

More recently, Hu et al. propose to use miss ratio curve for quantifying access locality and use the curve to determine the optimal space allocation for each class [9]. The optimality can be defined in terms of either hit ratio or average request service time. As miss penalty is included in the calculation of the average service time, the proposed scheme considers the aforementioned three factors. However, to enable effective application of dynamic programming technique, the scheme uses only *average* request hit time and *average* request miss time to quantify access time. We have shown in Figure 1 that miss penalty can vary in a large range. Average service time, or average miss penalty, measured in the previous time period may not be sufficiently representative and accurate to make the currently optimal allocation decision. In contrast, PAMA uses actual miss penalties associated with each slab to make the decision. Without tracking miss ratio curves and applying dynamic programming algorithm, PAMA makes use of individual miss penalties in the decision affordable.

III. Design of the PAMA Scheme

As we have indicated, PAMA is designed to have all three performance-critical factors, namely locality, size, and miss penalty, in a common cache management framework. To address the memory fragmentation issue, we follow Memcached's basic data structure, or use the slab as the allocation unit and place slabs into different classes, each for KV items in a certain range. In each class, items are managed with the LRU replacement policy. If there was only one class and all items were of the same miss penalty, the in-class LRU policy would exploit locality well, as long as relatively strong locality exists in the requests. With the existence of multiple classes, the critical issue becomes re-allocation of slabs between classes with the consideration of size and miss penalty. To this end, the three factors have to be quantitatively integrated into one measure, which is to be used as the basis for slab reallocation.

While locality cannot be directly quantified as size and miss penalty, we need to estimate its relative strength among items of similar size and miss penalty. To this end, we divide a class into a number of subclasses according to items' miss penalty. Items whose miss penalties are in a given range and that have been assigned to the same class are placed in the same subclass, and are organized as an LRU stack. By assigning KV items of different sizes into different classes and further assigning items of different miss penalties into different subclasses, each LRU stack (associated with a subclass) contains items of similar size and miss penalty and can be solely used for locality estimation, as shown in Figure 2. The bottom items in the stack (the LRU end of the queue) are of relatively weak locality among all items in the stack. If a slab has to be replaced from the subclass, due to misses either in this subclass or in another subclass, the bottom slab in the stack would be the one according to the locality distribution in the stack.

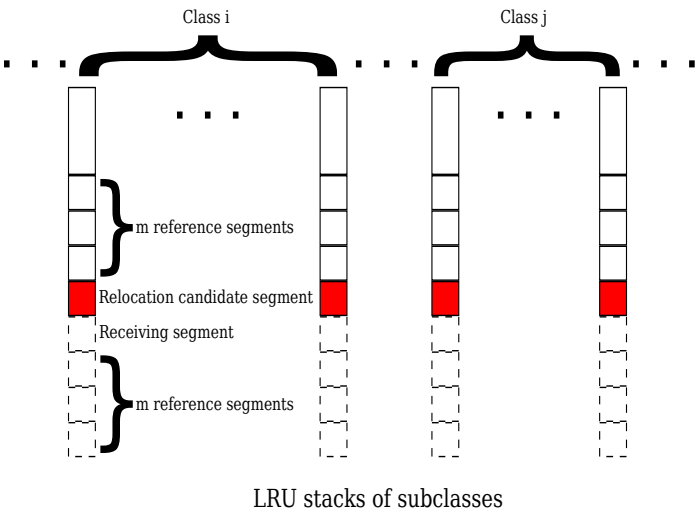

Fig. 2: Illustration of data structures facilitating PAMA's slab reallocation. A class is divided into a number of subclasses, each corresponding to a certain range of miss penalty values. KV items in each subclass are organized in an LRU stack. Relocation candidate segment indicates slab possibly for being taken away, and receiving segment indicates position where a new slab is placed in an LRU stack.

When a new slab is demanded in a subclass and no free slab is available in the cache, every subclass' bottom slab is a candidate for replacement to supply the requested slab space. Accordingly, the cache allocation problem becomes a selection among the candidates for minimizing request service time. The selection is guided by a candidate slab's value in a subclass, which is quantified as the amount of miss penalty that can incur when the slab is taken away from the subclass (*outgoing value*) during a given time window. A subclass also has its *incoming value*, which is the amount of miss penalty that can be saved when a new slab is added to the subclass to cache items recently replaced out of the subclass. In principle, when there is a miss in a subclass that does not have any free space, PAMA will potentially select the candidate slab of

the smallest outgoing value (among all subclasses' candidate slabs) for replacement. However, there are two scenarios where actual slab migration between (sub-)classes does not happen. First, if the incoming value of the subclass having the miss is smaller than the smallest outgoing value of the candidate slab, a migration does not help improve cache space utilization. Second, if the selected slab is from the subclass having the miss, there will be no cross-(sub)class slab migration, and one KV item in the slab will be be replaced.

To measure a candidate slab's value, we do not track accesses on a physical slab. Instead, it can be considered as a virtual slab holding items at the bottom of a subclass' stack. These items can physically reside in different physical slabs. When the virtual slab is to be replaced, the items are discarded in their respective physical slabs, and their left space will be reclaimed by moving valid items in the slabs together to produce an empty slab for migration. The time window in which the value is calculated is not wall-clock time. Instead, it refers to the number of accesses on the entire cache, as such a defined time period is more relevant to the cache replacement behaviors.

To effectively measure slab values, we need to address several challenges. First, values of individual slabs can be very dynamic in a relatively short time window. Using quickly changing values to determine slab reallocation can potentially lead to slab thrashing, or frequent moving slabs among a few subclasses. While increasing the time window helps stabilize the value, an excessively large time window makes the statistic less responsive to change of access patterns. Our approach is to take values of a number of slabs sitting above the candidate slab in a subclass's LRU stack into account. As shown in Figure 2, each of these slabs, including the candidate slab, in a subclass, corresponds to a segment of KV items in or near the bottom of the subclass's LRU stack. Each segment covers items that can be held in one slab. Assume that in the current time window, there are n requests $(R_1, R_2, ..., R_n)$ for items in a segment (S_k), where $k = 0, 1, ...$ The value of the segment, or the value of of its corresponding slab space, is

$$V_k = \sum_{i=1}^{n} T_i \tag{1}$$

where T_i is the miss penalty of the item requested by R_i. Assume a subclass's candidate slab is S_0 and slabs above S_0 are S_1, S_2, ... in this order. To calculate the candidate slab's value (V) for the purpose of slab migration, PAMA does not consider only V_0. Instead, it considers values of m additional reference segments close to it but gives higher weight to values of segments closer to the candidate slab.

$$V = \sum_{i=0}^{m} \frac{1}{2^{(i+1)}} V_i \tag{2}$$

This value is actually the outgoing value of the candidate slab, as it represents amount of performance loss for this subclass if the candidate slab is taken away from it.

The second challenge is how to estimate the incoming value for a subclass, or the amount of performance gain if an additional slab is added into the subclass. An added slab could turn some recent misses into hits. To know how many such misses could have been hits with the new slab, we extend the LRU stack beyond its current bottom to remember recently replaced items. As shown in Figure 2, this extended section only records keys and miss penalties of KV items, rather than the items' value components. If a new slab is received, it will be used to cache items in the segment right beneath the candidate slab in the LRU stack. This segment is called the receiving segment. To predict the incoming value of a slab in the subclass, it seems that we only need to calculate the miss penalties of items that had missed on the receiving segment in the current time window. However, for the same reason as that for considering multiple segments above the candidate slab, we consider m additional segments ($m \geq 1$) beneath the receiving segment in the LRU stack to calculate the incoming value. The method is similar. We use weighted values of the m reference segments, where higher weights are given to segments closer the candidate slab, to calculate the subclass' incoming value.

The third challenge is how to minimize the cost for calculating the number of hits or misses in each of the m segments. To know whether an access is a hit or a miss is equivalent to testing membership of an item in an item set. Scanning the segments upon each access is too expensive. Maintaining one or multiple hash tables for this purpose can lead to high space overhead, and updating membership in the hash tables needs lock(s) to serialize their access. To address this issue, we propose to use Bloom filters [1] to complete the testing in $\mathcal{O}(1)$ time with small space overhead. We use one Bloom filter for each reference segment. However, one can only add a member into a Bloom filter but cannot conveniently remove a member out of it. In the meantime, in the LRU stack a KV item has to be removed out of a segment once it is accessed. To address this issue, we set up a Bloom filter, called a removal filter, to track the items that have been recently removed out of the segments. When a segment's Bloom filter indicates that an item is in the segment, PAMA would consult with the removal filter. Only when the removal filter does not contain the item does PAMA consider the item to be in the segment. When PAMA detects that a new item being added into a segment is also in the removal filter, it clears the filter to ensure that the filter serves the purpose of indicating items not existent in the segments. Note that in the LRU policy, a removed item is placed at the top of the LRU stack and in a large cache storing many KV items it takes a substantially long time period for the item to re-enter the segments.

IV. Evaluation of the PAMA Scheme

To evaluate the performance of PAMA, we build a KV cache simulator that can faithfully simulate the behaviors of PAMA, as well as a hypothetical version PAMA, called pre-PAMA, that does not consider the miss penalty in the calculation of a segment's value. That is, in pre-PAMA a candidate slab's value is simply the number of requests in the segment. We use pre-PAMA as a reference scheme to demonstrate how miss penalty plays its role in improving the scheme's performance. Furthermore, we also simulate the original version of Memcached, where slab relocation among classes is not allowed, to evaluate the effect of the reallocation. As a representative of existing schemes that enable slab reallocation, PSA is also selected in the evaluation because it considers KV item sizes by using request density in the classes

and uses the density as the criterion to determine the source class of a relocation. In contrast, Facebook's Memcached policy [11] considers only item's age or access locality, and Twitter's Memcached policy [3] simply picks a random class as the relocation source. Therefore, we do not include them in the evaluation.

The workloads we use in the experiments are traces collected on Facebook's production Memcached cluster [6]. While the traces do not explicitly contain miss penalty information, we estimate it with the time gap between the miss of a GET request and the SET of the same key immediately following the SET. Most users add their missed KV items into a KV cache when they re-obtain them. We discard excessively large (larger than 5 seconds) time gaps as users may not immediately retrieve the KV items from the backup store or re-compute them at a database system, or they may not immediately add the items into the cache. For a small fraction of KV items whose miss penalties are unknown, we use a default penalty value (100ms), which is roughly the observed mean penalty, as their penalty.

The PAMA system in the evaluation follows the way in which Memcached defines its classes: the first class stores items of 64 bytes or smaller, the second class stores items of 128 bytes or smaller. In general, every class stores items whose maximum size doubles the one of its previous class. We divide a class into five subclasses, whose covered miss penalty ranges are (0, 1ms], (1ms, 10ms], (10ms, 100ms], (100ms, 1000ms], and (1s, 5s], respectively. To compute the value of a candidate segment, we use $m = 2$ reference segments, which we find sufficient to accurately quantify the value. We include a sensitivity study on the parameter at the end of the section.

The Facebook traces were collected from different Memcached clusters, each dedicated to serve requests from different services [6]. In total there are five clusters and five traces were collected. We choose two (ETC and APP) out of them in the evaluation. Among the other three traces (USR, SYS, and VAR), USR has two key size values (16B and 21B) and almost only one value size (2B). SYS has very small data set, and a 1G memory can produce almost a 100% hit ratio. VAR is dominated by update requests, such as SET and REPLACE, while we are mostly interested in GET requests whose performance is more related to the cache space management.

The performance metrics we use for reporting experiment results are hit ratio and average service time for GET requests. To obtain insights on the trend of these metric values' change during running of the workloads, we show their values in each time window (1 million GET requests). In the calculation of the metric values we only consider GET results, as they tend to impose high miss penalty and directly affect user-visible service quality.

A. Experiment with the ETC workload

We first experiment with the ETC trace, "*the most representative of large-scale, general-purpose KV stores*" [6]. As the trace has a relatively small data set, we start with a cache space of 4GB memory.

Let us see cache space distribution over the classes under the various schemes. Figure 3 shows how the space allocation among different classes changes over time. Without a space reallocation mechanism, the space allocation in Memcached does not change after all free memory has been allocated. Class 0 holds the smallest items but has the most frequent accesses. Accordingly, its allocation is ranked in about the middle of the list of the classes. Class 11, which holds the largest items, has the smallest memory allocation as its request rate is much lower than others'. Class 8 receives the largest allocation with its relatively modest item sizes and request rate. Under PSA, after the initial cache allocation Class 0 aggressively takes slabs from other classes. Class 0 receives over 70% of all requests to the system, and its request density is the highest among all classes until its allocation reaches 3.3GB. After this, all classes have about the same density and the allocation is stabilized.

Comparing PSA's allocation with the one produced by pre-PAMA, we can see that Class 0 also receives the largest share of cache space. However, pre-PAMA receives its allocation at a slower rate as the decision of reallocation is made according to the accesses on the segments around the bottom of an LRU stack. As PSA counts any accesses to make allocation decision, Class 0 may receive space from other classes because of many accesses on its non-near-bottom segments. As these accesses do not indicate immediate need of more space, in PSA Class 0 may receive more memory than what it needs. In contrast, in pre-PAMA, in addition to Class 0, some other classes, such as Classes 1, 2, and 3, can retain more of their spaces to reduce misses.

As seen in the Figure 3(d), the space allocation by PAMA is very different. The cache space is more evenly allocated across the classes. To find out the reason, we show the space allocation across subclasses within two example classes (Classes 0 and 8) in Figure 4. As shown, classes for small items, such as Class 0, have subclasses for items of small miss penalty, and can lose cache space. Meanwhile, classes for relatively large items, such as Class 8, have subclasses for items of relatively large miss penalty, may gain cache space. Therefore, consideration of miss penalty makes space difference between different classes' allocation become smaller.

The space allocation by PAMA is not necessarily optimal in terms of hit ratio. Figure 5 shows the hit ratio of the ETC workload with different cache sizes. As shown, PAMA's hit ratios are lower than those of PSA's, though their differences become smaller with a larger cache. By not caching some more frequently accessed but less expensive (in terms of miss penalty) items, PAMA may compromise hit ratio for the sake of minimal request service time. Pre-PAMA, which does not consider miss penalty and is optimized only for hit ratio, has the highest hit ratios. It achieves this by moving slabs to classes where the largest number of misses can be avoided. The original Memcached produces the lowest hit ratio, demonstrating a strong need of enabling slab relocation for higher performance. Another observation is that with a smaller cache, a scheme's hit ratio is more dynamic, as the caching behaviors are more sensitive to changes of access pattern. It is noted that using a larger cache increases hit ratio only by a few percentage points. This improvement is meaningful as it is reported that a mere 4.1% hit ratio increase "*represents over 120 million GET requests per day per server, with noticeable impact on service latency.*" [6].

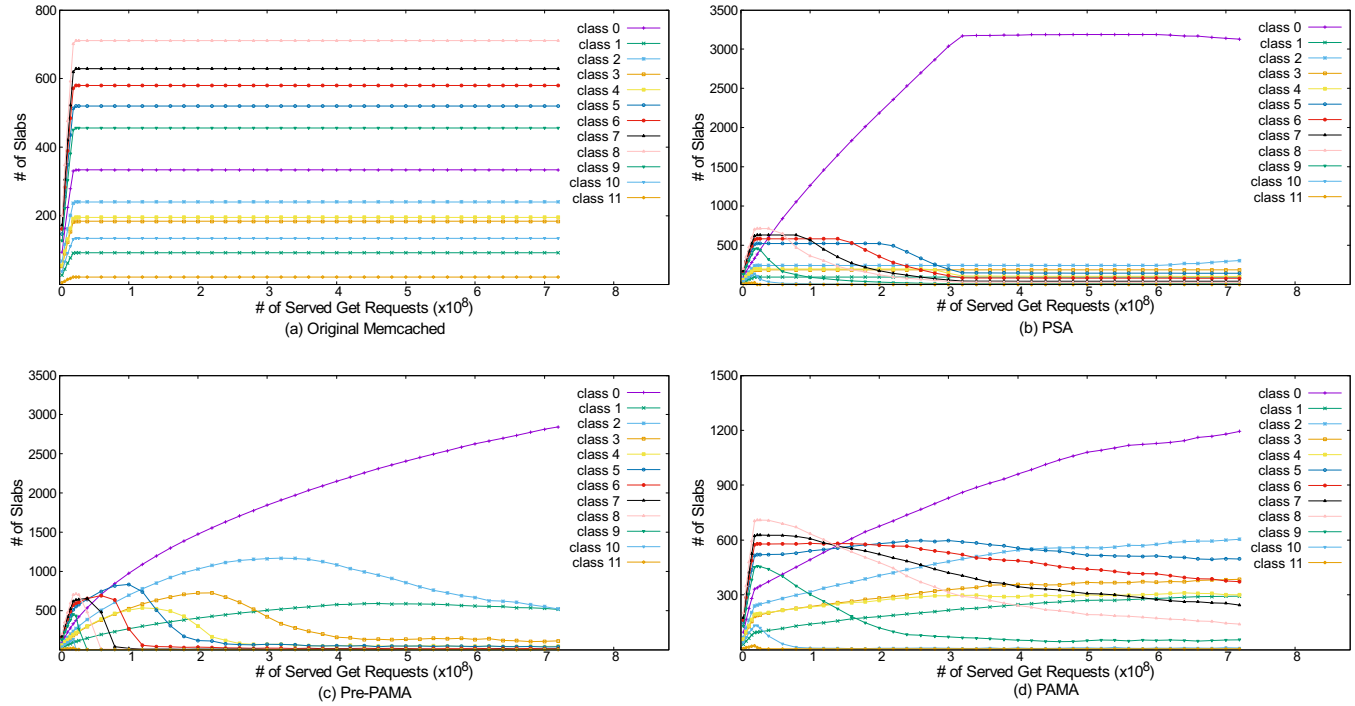

Fig. 3: Space allocations in different classes under various schemes in a 4GB cache (a) Memcached without space reallocation, (b) PSA, (c) pre-PAMA, and (d) PAMA.

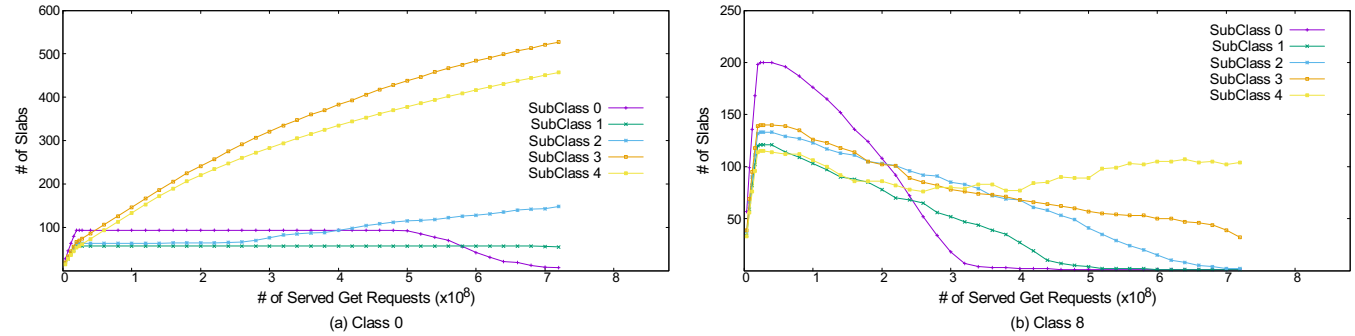

Fig. 4: Space allocations in different subclasses under the PSA schemes (a) in Class 0 and (b) in Class 8.

As we have stated, the ultimate metric for evaluating a KV cache's performance is request service time. Figure 6 shows average service times in KV caches of different sizes. Clearly in all cache sizes PAMA achieves the shortest service time. Though its hit ratios are higher than those of pre-PAMA and PSA, its service time can be substantially smaller, especially when the cache size is small. PAMA allocates cache space among classes and subclasses based on a slab's value, quantifying its contribution on reducing service time. When cache is relatively small, there are more misses and PAMA's service-time oriented optimization allows more misses to occur on items of relatively small miss penalty.

B. Experiment with the APP workload

We then experiment with the APP workload, one with a large data set in terms of aggregate accessed KV item sizes. Accordingly we use 16GB, 32GB, and 64GB as testing cache sizes. Because significant misses (around 40% of all misses) are cold misses, whose contribution on the hit ratio and service time is independent of cache management schemes, we repeat the same trace in the second half of the experiment to highlight the performance difference among the schemes.

Figure 7 shows the hit ratios under different schemes. Consistent with the observation on the ETC workload, Pre-PAMA achieves the highest hit ratios, and PAMA's hit ratios are even lower than those of PSA. Also with larger caches, dynamics of hit ratio curves become less dramatic because dynamics of changing access behaviors within short time periods can be better absorbed by additional cache space. Specifically, with more cache space, some misses on items that cannot be held in the cache can be turned into hits when additional cache space is made available. Another observation

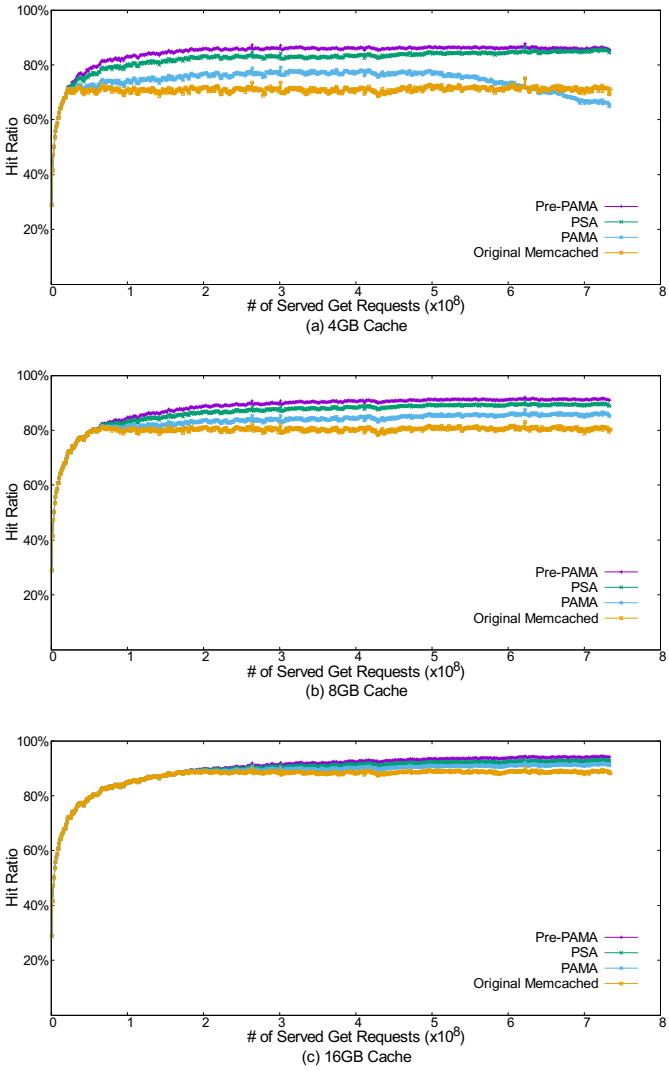

Fig. 5: Hit ratios of the ETC workload under various schemes with cache of different sizes: (a) 4GB, (b) 8GB, and (c) 16GB.

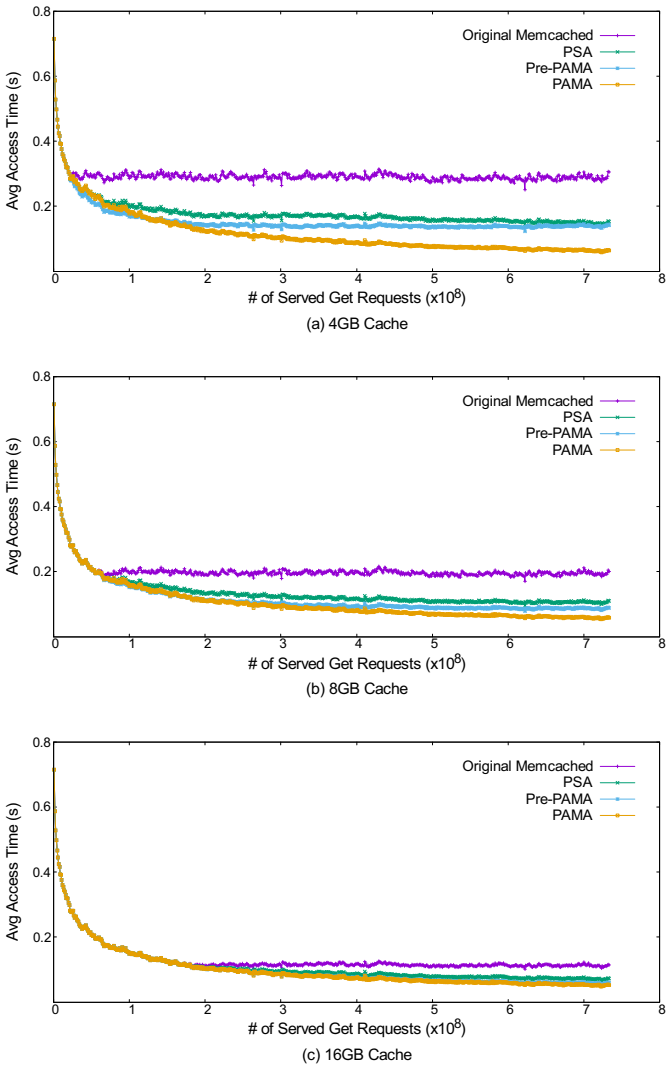

Fig. 6: Average request service time of the ETC workload under various schemes with cache of different sizes: (a) 4GB, (b) 8GB, and (c) 16GB.

is that the schemes that perform better in terms of hit ratio have smaller variations on the ratio, such as pre-PAMA and PSA, as they can avoid some of the suddenly increased misses with an optimized use of its cache space. Because in the second half of the experiment all cold misses do not exist, better performing schemes, such as pre-PAMA and PSA, have even higher improvements, clearly showing their advantages.

Figure 8 shows the workload's average request service time under different schemes. Consistent as what we saw in the ETC experiment, the relative advantage between PAMA and pre-PAMA/PSA is reversed when we change performance metric from hit ratio to service time. As shown in Figure 8, PAMA's reduction of service time is very impressive. Before removing cold misses, with a 16GB cache PAMA's average service time is only around 36% and 67% of the original Memcached's and PSA's times, respectively. When the service time curves stabilize, PAMA's service time is only 11% and 27% of Memcached and PSA's times, respectively, in the repeating trace that does not contain cold misses. Even with a much larger cache (64GB), the improvements are similar. The large improvements are expected considering the large variation of miss penalty (see Figure 1 and ignorance of miss penalty in other schemes' cache management.

C. *Study on the impact of caching unpopular items*

In a KV cache system, there can be a bursty stream of requests accessing and adding new KV items into the cache. The new items may be associated with hot data, which will be intensively accessed after they are cached, and accordingly caching them would help improve (at least maintain) cache performance in terms of hit ratio or average request service time. However, there are cases where the items are not (at least not immediately) popular ones. Because the LRU replacement policy is used in the caches, these relatively cold items will still be received into the cache. However, infrequent access of the cached items could compromise the performance. A caching scheme should minimize the impact and effectively respond to the access pattern.

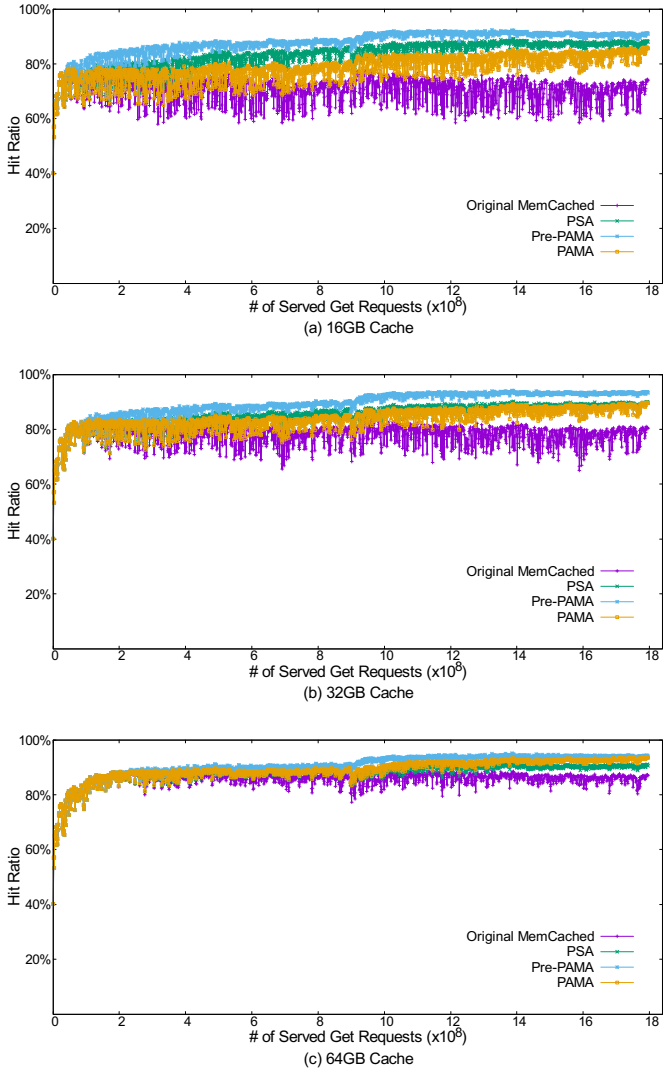

Fig. 7: Hit ratios of the APP workload under various schemes with cache of different sizes: (a) 16GB, (b) 32GB, and (c) 64GB.

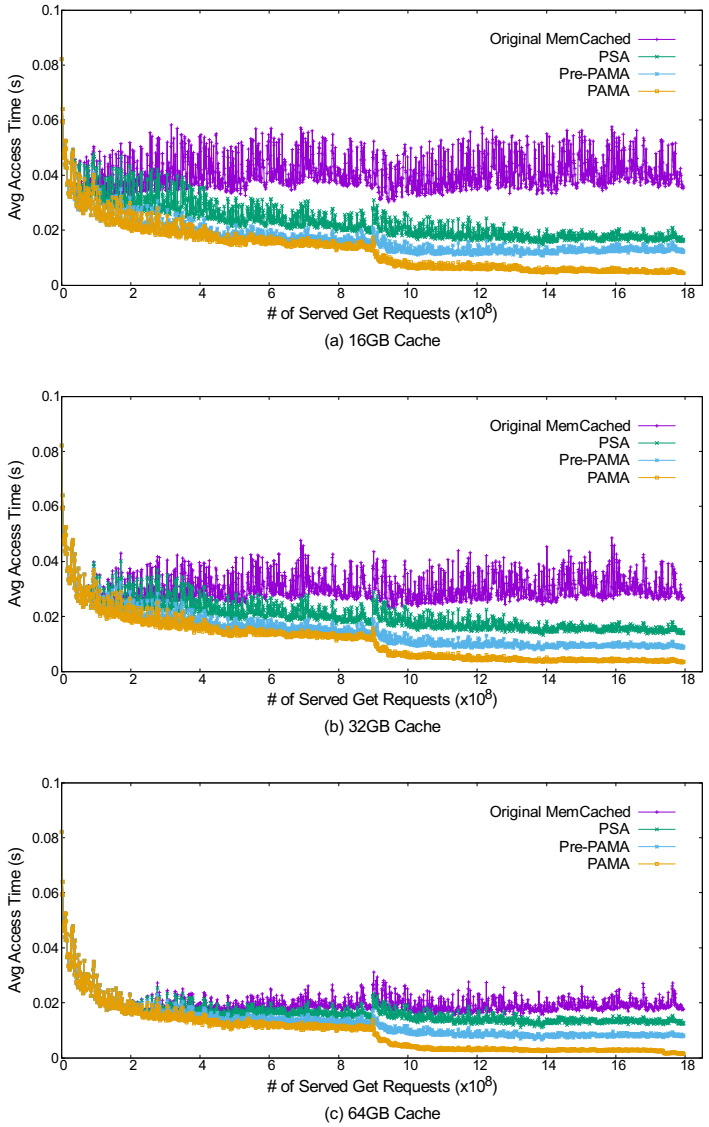

Fig. 8: Average request service time of the APP workload under various schemes with cache of different sizes: (a) 16GB, (b) 32GB, and (c) 64GB.

To gauge the impact, at the time of about 0.35 million GET requests we use the SET command to quickly inject cold KV items whose total size is about 10% of the cache size into a KV cache running the ETC workload. Because bursty requests are often from the same application or service and share some common characteristics, we limit the cold requests' sizes in a relatively small range covering only three classes, named as impacted classes. Figure 9 shows the impact of caching cold items on hit ratio and average service time under PSA and PAMA schemes. As shown in Figure 9(a), with the arrival of bursty cold items, PSA's hit ratio is accordingly reduced. This is well understood, as PSA always relocates slabs to the class(es) with the largest number of misses. The impacted classes receive the cold misses in a short time period and produce many misses. Accordingly they *steal* memory slabs from other classes and cause their hit ratios to reduce. However, because the misses in the impacted classes are due to unpopular items, the slabs received by these classes do not substantially lead to more hits. To make the situation even worse, after the impact of cold item access, it takes a long time period to recover the lowered hit ratio, as the impacted classes lose their less-efficiently used slabs slowly. Though the classes' request density becomes lower, their number of misses can also be high, as hot items may also have been evicted. Looking into Figure 9(b), we have similar observations on PSA's request service time: a sudden surge of the service time upon impact and a slow recovery from the impact.

In contrast, PAMA exhibits much more desirable behaviors. As shown in Figure 9(a), upon the impact, its hit ratio takes a small drop and recovers quickly. PAMA makes its decision on slab relocation based on the value of slabs at and near the bottom of a subclass's LRU stack, or their potential contributions to hit ratio or service time. When the cold items are added into the impacted classes and hot items in the classes are also accessed simultaneously, the cold items are pushed

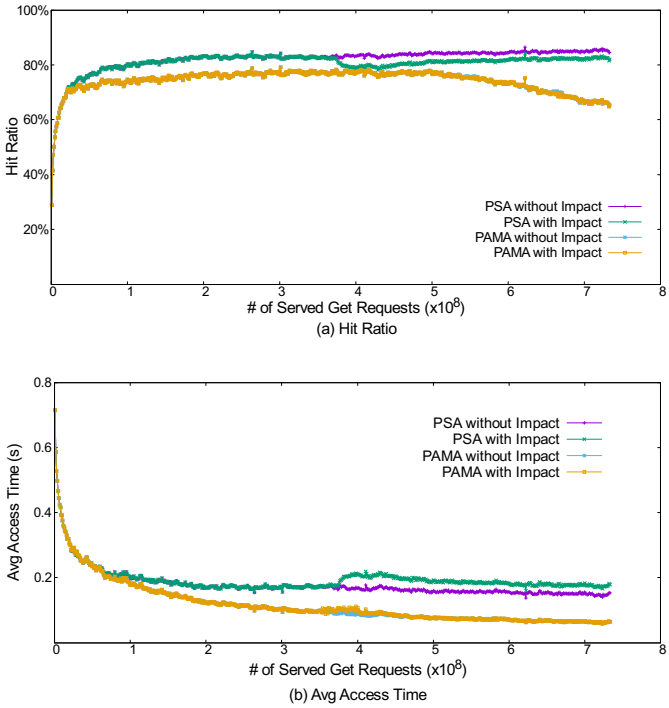

Fig. 9: Performance impact of adding unpopular KV items into a 4GB- cache serving the ETC workload. The performance metric can be either (a) hit ratio or (b) average request service time.

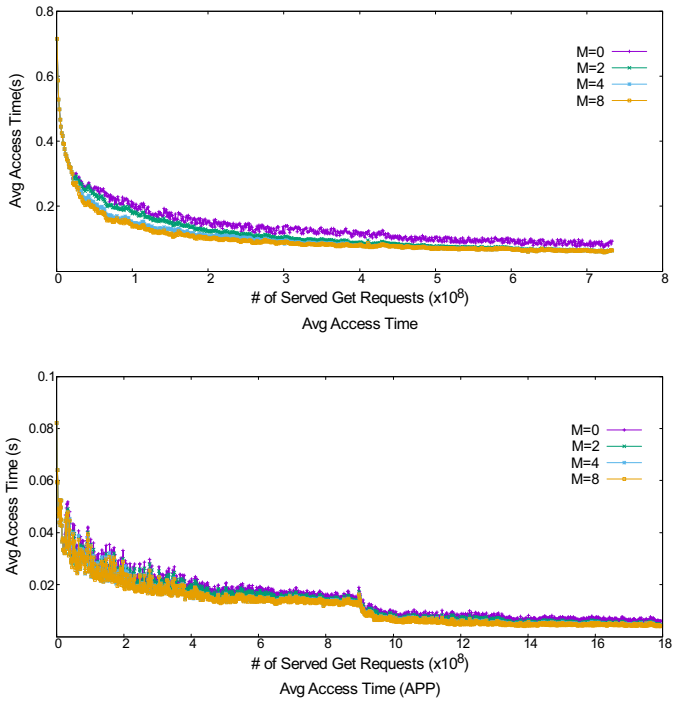

Fig. 10: Average request service time with different numbers of reference segments (M) in the PAMA scheme. (a) The ETC workload on a 4GB cache, and (b) the APP workload on a 16GB cache.

towards the stack bottoms, reducing the values of these classes' candidate slabs. This makes it harder for the classes to take others' slabs. For the same reason, after the impact, holding the cold items makes the classes more likely to lose slabs containing the cold items. PAMA's average request time, as shown in Figure 9(b), is little affected by the impact. The cache space relocation caused by accessing cold items happens mostly on low-miss-penalty slabs, as they are more likely to be selected for relocation. This makes the increased misses due to the impact be mostly 'less expensive' ones, leaving the impact on request service time smaller.

D. Sensitivity Study on PAMA Parameter

The PAMA scheme has a parameter, m, about the number of reference segments included in the calculation of a candidate slab's value. It is noted that in PAMA calculation of the value is essentially a prediction of a slab's future contribution to the cache's performance. A large m value helps alleviate effect of short-term access behaviors on the value and makes the value more indicative of future access characteristics, or makes it represent a more accurate prediction. To evaluate the parameter's impact on PAMA's performance, we vary m and see how the average request service time changes accodingly. As shown in Figure 10, when we increase m from 0 to 2, ETC's service time reduces by about 12-28%. Further increasing it to 4 and 8 brings a small additional time reduction. Comparatively, the impact of larger m on APP's service is visible but at a smaller scale. As very large m is apparently not necessary, choosing a moderate one, such as the default m value (2) used in the rest of the evaluation, is needed, especially for KV caches with more dynamical or less predictable workloads.

V. Conclusion

KV caches have been a critical infrastructure component for today's Internet-wide computing at data centers. While data caching has been studied for decades in various system levels, the workload characteristics of the cache for KV items are unique and demand new solutions. These characteristics include (1) KV item size can vary in a large range, as a vast variety of applications or services may share the same cache. (2) A KV item's miss penalty can vary in a large range as the items may be obtained by computing in back-end database systems. (3) The three important factors (locality, size, and miss penalty) that determine a KV cache's performance are not necessarily correlated. For example, a small item may have large miss penalty, and a frequently accessed item may have a small miss penalty. The PAMA management scheme proposed in this paper considers these three workload characteristics and integrates the three factors seamlessly in a common framework for dynamic space reallocation.

PAMA uses the slab as a reallocation unit, and uses item classes and subclasses to accommodate requirements on distinction of size and miss penalty, respectively. It uniquely introduces the concept of a slab's value to quantify the effect of KV items' locality, size, and miss penalty on the cache's performance, and uses it as the metric for space reallocation. Our experimentation with real-world Memcached workload traces demonstrates that PAMA can significantly reduce request service time compared with representative KV cache

management schemes. With the existence of highly variable miss penalties, existing KV cache management schemes that use hit ratio as the performance metric can lead to suboptimal performance experienced by users. In the evaluation, PAMA may not provide the best hit ratio. Instead, it is designed to optimize the most meaningful performance metric – request service time.

ACKNOWLEDGMENT

This work was supported by US National Science Foundation under CAREER CCF 0845711 and CNS 1217948. We thank Facebook Inc and Eitan Frachtenberg for their donating servers and sharing Memcached traces, which allowed two of the authors (Yuehai Xu and Song Jiang) to conduct extensive experiments for the evaluation. We thank the anonymous reviewers, who helped improve the quality of the paper substantially.

REFERENCES

[1] Bloom filter, *http://en.wikipedia.org/wiki/Bloom_filter*.

[2] D. Carra and P. Michiardi, "Memory Partitioning in Memcached: An Experimental Performance Analysis", In *Proceedings of IEEE International Conference onommunications*, 2014

[3] Caching with twemcache. In *https://blog.twitter.com/2012/caching-with-twemcache*, 2014.

[4] Memcached. *http://memcached.org/*

[5] Memcached-1.4.11. In *https://code.google.com/p/memcached/wiki/ReleaseNotes1411*, 2014.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store", In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* 2012.

[7] B. Fan, D. G Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing", In *USENIX Symposium on Networked Systems Design and Implementation*, 2013.

[8] B. Fitzpatrick, "Distributed caching with memcached", In *Linux journal*, 2004(124):5, 2004.

[9] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache", In *Proceedings of 2015 USENIX Annual Technical Conference*, 2015.

[10] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an Effective Buffer Cache Management Scheme to Exploit both Temporal and Spatial Locality", In *Proceedings of the 4th USENIX Conference on File and Storage Technologies*, 2005.

[11] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook", In *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*, 2013.