International Journal of Parallel Programming (2019) 47:451–466
https://doi.org/10.1007/s10766-018-0616-4

Lightweight and Accurate Memory Allocation in Key-Value Cache

Cheng Pan[1,2] · Lan Zhou[1] · Yingwei Luo[1,2] · Xiaolin Wang[1] · Zhenlin Wang[3]

Received: 20 September 2018 / Accepted: 16 November 2018 / Published online: 3 December 2018

Abstract

The use of key-value caches in modern web servers is becoming more and more ubiquitous. Representatively, Memcached as a widely used key-value cache system, originally intended for speeding up dynamic web applications by alleviating database load. One of the key factors affecting the performance of Memcached is the memory allocation among different item classes. How to obtain the most efficient partitioning scheme with low time and space consumption is a focus of attention. In this paper, we propose a lightweight and accurate memory allocation scheme in Memcached, by sampling access patterns, analyzing data locality, and reassigning the memory space. One early study on optimizing memory allocation is LAMA, which uses footprint-based MRC to optimize memory allocation in Memcached. However, LAMA does not model deletion operations in Memcached and its spatial overhead is quite large. We propose a method that consumes only 3% of LAMA space and can handle read, write and deletion operations. Moreover, evaluation results show that the average stable-state miss ratio is reduced by 15.0% and the average stable-state response time is reduced by 12.3% when comparing our method to LAMA.

✉ Cheng Pan
pancheng@pku.edu.cn

Lan Zhou
lanzhou@pku.edu.cn

Yingwei Luo
lyw@pku.edu.cn

Xiaolin Wang
wxl@pku.edu.cn

Zhenlin Wang
zlwang@mtu.edu

1 Department of Computer Science, Institute of Network Computing and Information Systems, Peking University, Beijing, China

2 Shenzhen Key Lab for Information Centric Networking and Blockchain Technology (ICNLAB), SECE, Peking University, Shenzhen, China

3 Department of Computer Science, Michigan Technological University, Houghton, MI, USA

Keywords Key-value cache · Lightweight and accurate memory allocation · Performance prediction and optimization

1 Introduction

Caching has been applied to all levels of the memory/storage architecture, especially for in-memory key-value stores. In today's web server architecture, distributed in-memory caches, such as Redis [1] and Memcached [2], are vital components to ensure low-latency service for user requests. Memcached as a most commonly used distributed memory key-value cache system, has been deployed in Facebook, Twitter, Wikipedia, Flickr, and many other Internet companies. Given the scale and scope of these deployments it has been shown that even a slight improvement in hit-ratio can have a significant impact on performance [3,4], so any improvement in hit-ratio is prominent in realistic scenario.

By default, items in Memcached are divided into several data classes according to the size of the item, and the memory is allocated according to the request, regardless of the locality of the data. This demand-driven allocation algorithm does not achieve the desired performance. This problem is called *slab calcification* [5]: When all available slabs are allocated, the allocated space can no longer be adjusted to adapt the changed access pattern.

Memcached allocates space at slab granularity, which is 1 MB by default. For each class, the allocated objects are filled in the slab to which they are assigned. These objects are sorted based on their last access time to form an LRU linked list in a priority queue. If there is no free space in the class, a data item will be evicted following the LRU policy. In this design, the number of slabs in each class represents the memory space that has been allocated to it.

Performance prediction [6,7] and optimization [8–12] for Memcached have drawn much attention recently. To avoid the performance drop due to *slab calcification*, the operator needs to restart the server to reset the system. Recent studies have proposed adaptive slab allocation strategies and shown a notable improvement over the default allocation [13,14]. A state-of-art solution, *locality-aware memory allocation* (LAMA) [15,16], is a novel dynamic slab allocation scheme. It applies the *footprint* theory [17] to construct miss ratio curves (MRCs) for each class in Memcached, and then uses dynamic programming to determine how many slabs are needed for each class. LAMA's spatial overhead is linear to the working set size. For 1 GB working set, LAMA needs to consume nearly 100 MB of space to construct MRC, which is quite large especially when we running LAMA for large applications.

Moreover, deletion operation is a common Memcached operation that LAMA can not model. In the Memcached implemented by Facebook [14], in order to avoid inconsistent data (e.g. concurrent update operations, may cause inconsistencies between Memcached and the back-end database), when one client needs to update data, it will delete the corresponding data from Memcached and only update the back-end database. Figure 1 is Memcached's operations in Facebook, as shown in Fig. 1b, we first update the contents of the database, and then delete the corresponding key in Memcached. For more general scenarios, we need to enhance LAMA.

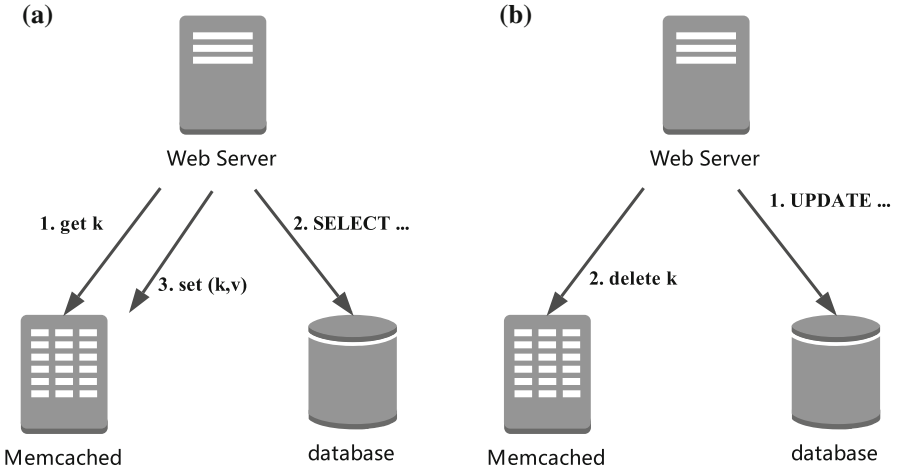

Fig. 1 Memcached operations in Facebook. **a** Get operation and **b** update operation

In this paper, we propose a *lightweight and accurate memory allocation* scheme in key-value cache, based on an extension of a recent cache model, AET [18]. We use the extension model to achieve high accuracy on miss ratio with read, write and deletion operations. In addition, we amend the cost function of the dynamic programming in LAMA, making the cost function closer to the real miss count during phase changes in Memcached. We compare our solution to LAMA, the spatial overhead is reduced to only 3.0% of LAMA. In addition, the average stable-state miss ratio can be reduced by 15.0% and the average stable-state response time can be reduced by 12.3%.

2 MRC Construction

Miss ratio curves (MRCs) are useful for estimating *how much* data is being used by a particular workload and *what utility* can be gained by increasing the memory size [19]. We allocate memory to each class by measuring the MRCs to each class in Memcached. We firstly review the *footprint* in LAMA and AET model for an LRU cache, and then describe an *Enhanced AET* (*EAET*) to model deletion operations.

2.1 Footprint in LAMA for MRC Construction

In LAMA design, the global access trace will be splitted into different sub-traces according to their classes. With the sub-trace of each class, the MRCs can be generated using *footprint* theory.

LAMA uses a hash table to record the *last access time* of each item. With this hash table, LAMA can easily compute the reuse time distribution $rt(i)$, which represents the number of accesses with a reuse time i. Reuse time is the time interval between two accesses to the same object. It is a simpler metric to measure locality than reuse distance, which counts the number of distinct accesses between an access and its next

reuse. For access trace of length N, if the number of unique data is M, the average number of items accessed in a time window of size w can be calculated using Xiang's formula [17]:

$$fp(w) = M - \frac{1}{N-w+1}\left(\sum_{i=1}^{M}(f_i - w)I(f_i > w) + \sum_{i=1}^{M}(l_i - w)I(l_i > w) + \sum_{t=w+1}^{N-1}(t-w)rt(t)\right). \tag{1}$$

Here, a time window of size w is a sub-sequence with length w of consecutive accesses in the trace. In Eq. 1, f_i is the first access time of the ith datum, l_i is the *reverse* last access time of the ith datum. If the last access is at position x, $l_i = n + 1 - x$, that is, the first access time in the reverse trace. $I(p)$ is the predicate function equals to 1 if p is true, otherwise 0. $rt(t)$ represents the number of accesses with a reuse time t.

Finally, LAMA can profile the MRC using fp distribution. The miss ratio for a cache of size c is the fraction of reuses that have an average footprint larger than c:

$$mr(c) = 1 - \frac{\sum_{\{t|fp(t)<c\}} rt(t)}{N} \tag{2}$$

In LAMA, the sub-trace of each class need to be stored to calculate the f_i, l_i and $rt(t)$, this leads to a linear relationship between the consumption of space and the length of access trace, which may consume too much memory when the working set size is large.

2.2 AET Modeling

The AET model is a series of kinetic equations related to average data eviction in the cache [18,20]. The only input to the AET model is *Reuse Time Histogram (RTH)*, and the output is the *Miss Ratio Curve (MRC)*.

With RTH, we can now calculate the probability that reference x has reuse time greater than t , which is then related to a stack movement in an LRU queue.

The AET model establishes a relationship between the reuse time distribution and the *average eviction time (AET)*. AET zooms in on the *eviction process* of an *evicting block* from its last access to its eviction. During this process, the evicting block spends eviction time, $AET(c)$, on average, to travel c stack positions from top to bottom and then get evicted. It is realized by the following equation:

$$\int_0^{AET(c)} v(t)\mathrm{d}t = c. \tag{3}$$

Here, $v(t)$ is the traveling speed of the data blocks, at time t. $AET(c)$ represents the time a data item travels from top to bottom of the LRU stack with size c. Hu et al. show that $v(t)$ is equal to $P(t)$, the probability that a reference has reuse time greater than t. Now Eq. 3 turns to:

$$\int_0^{AET(c)} P(t)\mathrm{d}t = c \tag{4}$$

We can obtain $P(t)$ from the reuse time histogram:

$$P(t) = \sum_{i=t+1}^{\infty} \frac{rt(i)}{N}, \tag{5}$$

where N is the length of the sampled sequence and $rt(i)$ is the number of accesses with reuse time i.

Finally, the miss ratio $mr(c)$ for an LRU cache with size c is indeed the probability that a reuse time is greater than the average eviction time:

$$mr(c) = P(AET(c)) \tag{6}$$

When the reuse time distribution is sampled, Eq. 4 finds $AET(c)$ and then Eq. 6 gives the miss ratio curve, i.e., $mr(c)$ for all c, which can be efficiently calculated in linear time.

2.3 Enhanced AET Modeling

The AET model is a primary model for cache read and write operations, without regarding to deletion operations. A recent work *PACE* [21] comes up with a more general scenario on an LRU cache, which proposes a method to be able to model various operations such as read, write, update and deletion on data items with different sizes.

In Memcached, we only need to consider the MRC for each class, and each item in a class occupies a same size slot, so we only need to consider set, get and deletion operations. Our enhanced AET considers possible upward data movements due to deletions in an LRU stack. The movement of a data block can be bi-directional:

1. An downward move, caused by a set or get operation. (sample reuse time distribution in $access\_rth[i]$);
2. An upward move, caused by a deletion operation. (sample reuse time distribution in $delete\_rth[i]$).

Assume that we have the reuse time distribution of all the read, write and update operations to any data item as well as the delete time distribution of all deletions. Assume $fa(i)$ presents the probability that an item is accessed (read, written or updated) with a reuse time i, $fd(i)$ presents the probability that an item is deleted with a delete time i. For each operation, the probability for an item of data with arrival time t to move one step towards the stack bottom is $P(t)$:

$$P(t) = \sum_{i=t+1}^{\infty} fa(i) - \sum_{i=0}^{t-1} fd(i) \tag{7}$$

With the reuse time distribution of access and delete, we can easily find *fa* and *fd*:

$$fa(t) = \frac{access\_rth[t]}{N} \tag{8}$$

$$fd(t) = \frac{delete\_rth[t]}{N} \tag{9}$$

The overall speed of movement ($v(t)$, $or\, P(t)$) is just the superposition of the velocities in both directions:

$$P(t) = \sum_{i=t+1}^{\infty} \frac{access\_rth[i]}{N} - \sum_{i=0}^{t-1} \frac{delete\_rth[i]}{N} \tag{10}$$

Here, $\sum_{i=t+1}^{\infty} \frac{access\_rth[i]}{N}$ is the downward movement probability (or speed) caused by access operations (set/get), and $\sum_{i=0}^{t-1} \frac{delete\_rth[i]}{N}$ is caused by deletion operations. It is conceivable that if the deletion speed is greater than the access speed, the data block in the LRU stack tends to move upward as a whole. With the travel speed set (i.e. $P(t)$), Eq. 4 can be applied to find the average eviction time ($AET(c)$) and Eq. 6 derives the miss ratio curve.

3 Lightweight and Accurate Memory Allocation

Many real-world workloads change their access patterns over time, in order to achieve high performance, we need to dynamically reassign memory of some classes in face of changing workloads. We adopt the idea of dynamic programming to reassign the slabs of each class in LAMA and use EAET to construct MRCs for each class. The whole process of our memory allocation scheme is shown in Fig. 2, and we also amend the cost function when doing dynamic programming, which will be explained in Sect. 3.2.

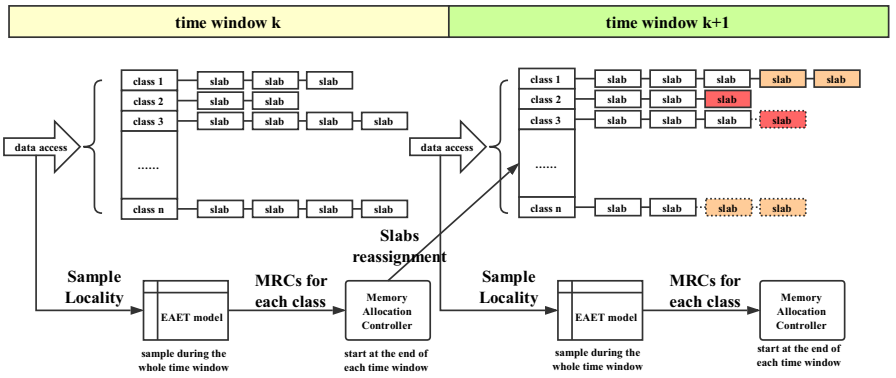

Fig. 2 Process of lightweight and accurate memory allocation

3.1 Memory Allocation Controller in LAMA

In LAMA design, a workload can be divided into a series of fixed-size time windows (or phases). In a time window, it collects data accesses within the window to build the MRC, calculates the best allocation scheme to achieve minimal total miss rate or average response time, and finally apply this allocation scheme to next time window. The size of time window can be set by the user or empirically.

For instance, if we allocate class i with M_i memory, then this class's overall performance target (miss rate or response time) PT_i can be calculated as:

$$PT_i = cost_i(mr_i(M_i), N_i) \tag{11}$$

where N_i is request number of class i, and the miss ratio is $mr_i(M_i)$, which can be derived from the MRC. $cost_i$ is the cost function of the performance target:

$$cost_i(m, cnt) = \begin{cases} cnt * m, & target = miss\_rate \\ cnt * (m * T_m(i) + (1-m) * T_h(i)), & target = response\_time \end{cases} \tag{12}$$

Here, $T_m(i)$ is the average miss time of class i, $T_h(i)$ is the average hit time of class i. Our final goal is to optimize the overall performance for all classes:

$$min \sum_{i=1}^{C} PT_i = min \sum_{i=1}^{C} cost(mr_i(M_i), N_i) \tag{13}$$

$$s.t. \sum_{i=1}^{C} M_i = M \tag{14}$$

where C is the number of classes, M is total memory we can allocate. The optimal solution can be reached through dynamic programming (see Algorithm 1).

Algorithm 1 Dynamic Programming for Memory Allocation

```
for i ← 1..C do
    for j ← i + 1..M do
        f[i][j] ← ∞
        for k ← i..j do
            f[i][j] ← min(f[i][j], f[i − 1][k] + cost_i(mr_i(j − k), N_i))
        end for
    end for
end for
```

The most important part of this dynamic program is the target function:

$$f[i][j] \leftarrow min(f[i][j], f[i-1][k] + cost_i(mr_i(j-k), N_i)) \tag{15}$$

where $f[i][j]$ represents the optimal performance when the first i classes are allocated j slabs. This is an intuitive target function, but there is a subtle deviation in the cost function, which we will discuss next.

3.2 Correction of Cost Function

Our memory allocation scheme changes phase by phase. In phase t, the amount of memory each class being allocated is $slabs[i]$. At the end of phase t, we will perform the slabs reassignment using dynamic programming. The reassignment scheme can be expressed as $next\_slabs[i]$, which means in phase $t+1$, the amount of memory allocated to each class is $next\_slabs[i]$. In stable stage, we always have $\sum slabs[i] = \sum next\_slabs[i]$.

According to the previous target function (i.e. Eq. 15), the expected total number of misses in phase $t+1$ is: $mr_i(next\_slabs[i]) * N_i$. But in fact, transitioning from $slabs[i]$ to $next\_slabs[i]$ is not a cold start, while the footprint and EAET model all cold misses, which can overestimate the number of misses.

Each memory reassignment decision takes place in two adjacent phases. We use $memory[t]$ denote the memory allocated to a specific class in phase t, and $memory[t+1]$ denotes the memory allocated to the same class in phase $t+1$. We intercept the two phases in the run to test this deviation, Fig. 3 shows the ratio between the real miss count in phase $t+1$ and estimated miss count using the cost function at the end of phase t. Here, the real miss count is achieved by running LAMA multiple times under different memory sizes and counting the real misses in phase $t+1$. In Fig. 3a we can see the deviation is quite a big problem. The vertical axis is the ratio between real miss count and estimated miss count. The horizontal axis is the ratio between $memory[t+1]$ and $memory[t]$. The ideal situation is that all ratios are 1.

The key factor that leads to this deviation is: *We assume that each phase is cold-started*. The EAET model and the footprint model both assume that the cache is cold-started. However, the real situation is that the reassignment between two phases

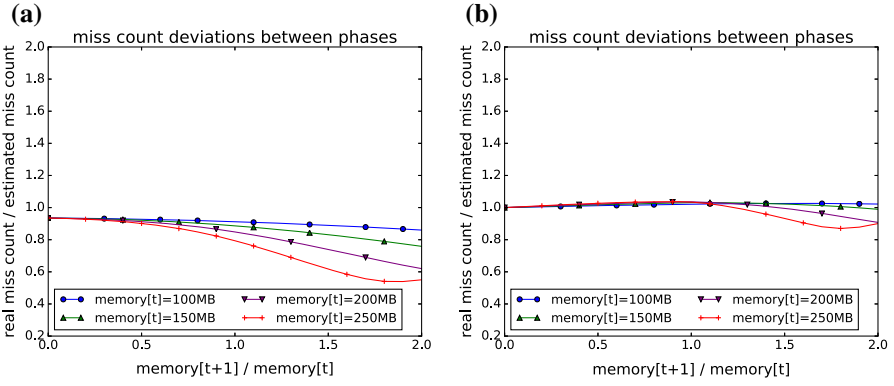

Fig. 3 Miss count deviations. **a** Before amendment, **b** after amendment

is not a cold start. Before each reassignment starts, there is already a part of data in each class.

We can do some amendments by comparing the size relationship between $memory[t+1]$ and $memory[t]$:

1. $memory[t+1] < memory[t]$ At this point, the memory used in phase $t+1$ becomes smaller than phase t. When we collect RTH, we ignore the cold miss, and capture long reuse time across phases.
2. $memory[t+1] \geq memory[t]$ At this point, the memory allocated in phase $t+1$ becomes bigger than phase t. Because a class's expansion is a gradual process, so we need to fit this gradual process to get better accuracy. In the AET model, the logical time required to fill a cache of size c is $AET(c)$, that is, we need to spend $AET(memory[t+1]) - AET(memory[t])$ logic accesses to transit from $memory[t]$ to $memory[t+1]$, and the misses due to this reason is linearly related to the expanded memory allocation. After the filling process, the stabilization situation is reached and the miss rate is $mr_i(memory[t+1])$. We can divide the cost function into two segments, one for gradual conditions, and the other for stable conditions.

For example, we amend the cost function when the target performance is the miss rate:

$$cost_i(slab_1, slab_2, cnt) = \begin{cases} cnt * mr_i(slab_2) - cold\_miss_i & slab_1 \geq slab_2, \\ (cnt - (AET_i(slab_2) - AET_i(slab_1)) * & \\ mr_i(slab_2) + (slab_2 - slab_1) * IPS_i & slab_1 < slab_2 \end{cases}$$

Here, the cost function is used to estimate the miss count in the next phase, the arguments include $slab_1, slab_2, cnt$, which means the slabs used in phase t is $slab_1$ and the allocated slabs in next phase is $slab_2$, the total request number is cnt. The subscript i indicates a particular class i, and IPS_i denotes *data items per slab in class* i. When the target performance is average response time, it's very easy to extend:

$$\begin{aligned} average\_response\_time_i = & T_m(i) * cost_i(slab_1, slab_2, cnt) + \\ & T_h(i) * (cnt - cost_i(slab_1, slab_2, cnt)) \end{aligned} \tag{16}$$

In Fig. 3b, we illustrate the effect of amendments on cost function. Obviously, these lines are more closer to the standard line (i.e. ratio = 1) .

4 Evaluation

In order to clearly evaluate the different MRC measurement methods and the effect of adding the cost function amendment, we have split five different variants:

LAMA-fp The original LAMA scheme.
LAMA-aet Only use AET model do MRC Construction.
LAMA-eaet Only use EAET model do MRC Construction.

LAMA-aet-amend Use AET model and the amendment of cost function.
LAMA-eaet-amend Use EAET model and the amendment of cost function.

4.1 Environment Setup

LAMA Configuration In the implementation of LAMA-fp, the phase is divided according to the re-partition interval M, and update the hash table independently at each phase, record the $reuse\_time$, $last\_visit\_time$, $first\_visit\_time$, and then calculate the MRCs of this phase. Because the MRC calculation is in a separate background thread (lru maintainer thread), the MRC calculation time can be ignored. There are two parameters that need to be set in the LAMA, namely, the re-partition interval M, and the upper bound N of the number of reassigned slabs in one adjustment. Here M is set to 10 million and N is set to 200.

System Configuration The machine we used for the experiment was Intel(R) Core(TM) i7-3770, 4-core, 3.4 GHz, 8 MB LLC and 16 GB memory. In the experiments, we measured miss rate and response time. To measure the latter, we set up a MySQL database for back-end storage for Memcached. The response time includes the time when the data was retrieved from the back-end database when the data did not hit. In our experiments, the Memcached instance was running on the local port with 4 threads, and the database was running on another server in the same LAN.

Workloads We use two different traces to evaluate the performance of variants of LAMA: Facebook ETC trace and a Redis trace from a cloud computing service provider. The Redis trace is transformed into a sequence of operations suitable for Memcached and it originally contains deletions operations. Facebook ETC trace is generated using Mutilate [22]. Mutilate mimics the features of Facebook's ETC workload. The ETC workload is the closest to the general workloads and has the highest miss rate in all Facebook Memcached pools. The Facebook ETC trace has 90 million requests and 7 million data objects and the Redis trace has 160 millions requests in all.

4.2 Impacts of Deletion Operations

Because the original LAMA did not consider the modeling of deletion operations, we evaluate how much this deviation would be in a scenario with a real deletion operation. In Facebook ETC, we set the ratio of the update operation and get operation to 1:9, and follow the operation processing of Facebook in Fig. 1, turn some of the update

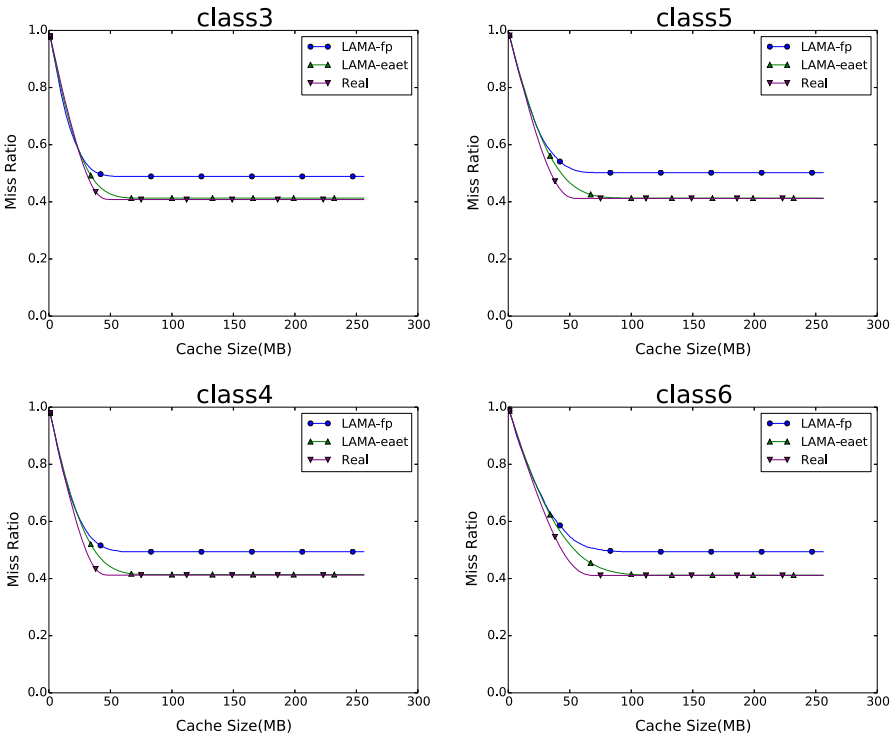

Fig. 4 MRC in different classes

operations into deletion operations, the ratio of the deletion operation to the update operation is subject to the Zipf($c = 1.0, \alpha = 2.0$) distribution.

In Fig. 4, we show MRC predictions for data class 3–6, and other classes have the same situation. It can be easily found that the LAMA-eaet is very close to the real MRC, and because the LAMA-fp does not take deletion operations into account, the predicted MRC results are quite different from the real MRC. Overall, the average absolute error of LAMA-eaet is 0.7%, while the average absolute error of LAMA-fp is 7.3%.

4.3 Total Miss Rate and Average Response Time

In this section, we will evaluate two different target performances: total miss rate (MR) and average response time (ART). The difference between the two is reflected in the difference of the cost function in Eq. 12.

Facebook ETC and a cloud computing trace are used to evaluate the performance, and we set the memory limitation to $1/2$ of the working set size. We show the evaluation of miss rate and average response time in Fig. 5. The left half shows the miss rate, and the right half shows the average response time.

We can find, LAMA-fp's performance on both indicators is worse than ours, and the amendment of the cost function does have an effect. In Fig. 5a, the average miss

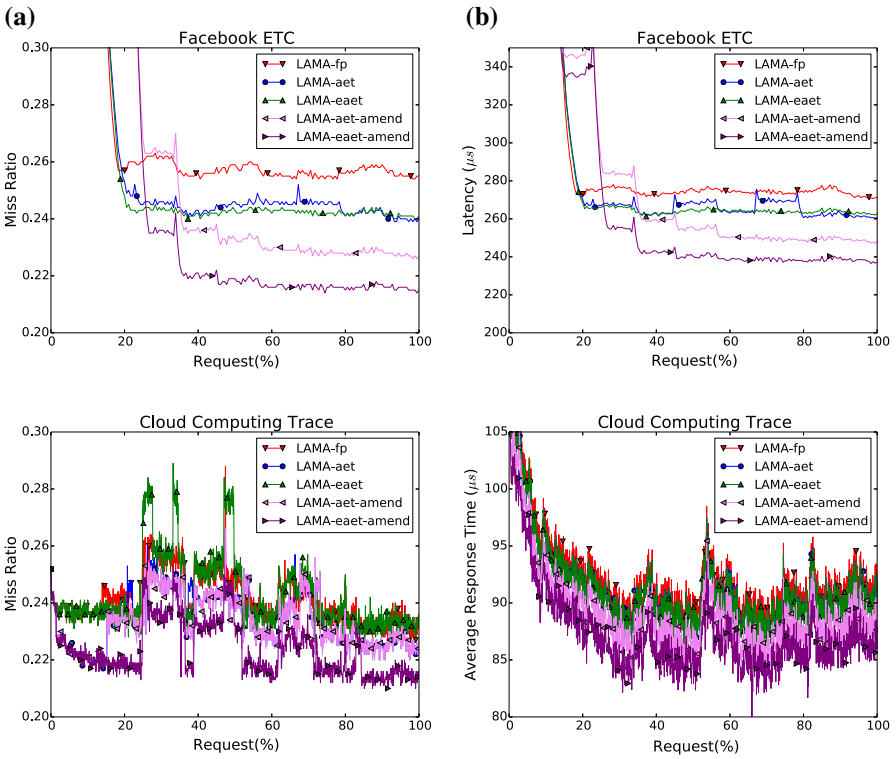

Fig. 5 Target performance on different variants. **a** Miss rate and **b** average response time

rate is reduced by 15.6% (from 25.6 to 21.6%) in Facebook ETC and 7.8% (from 23.3 to 21.5%) in the cloud computing trace when comparing LAMA-eaet-amend to LAMA-fp.

In Fig. 5b, the ART situation is similar to MR. The average response time is reduced by 13.0% (from 274.1–238.5 μs) in Facebook ETC trace and 5.4% (from 91.7 to 86.7 μs) in the cloud computing trace when comparing LAMA-eaet-amend to LAMA-fp.

We also evaluate the stable-state performance for memory sizes from 128 to 1024 MB in 128 MB increments. In Fig. 6, LAMA-eaet-amend always outperforms than LAMA-fp in terms of miss rate and average response time. Comparing LAMA-eaet-amend and LAMA-fp, the average stable-state miss ratio is reduced by 15.0% (9.8–18.2%) and the average stable-state response time is reduced by 12.3% (7.5–15.3%).

4.4 Tail Latency

In this part, we evaluate the tail latency of LAMA-eaet-amend, LAMA-fp and Automove using Facebook ETC trace. Automove is the default reassignment strategy used

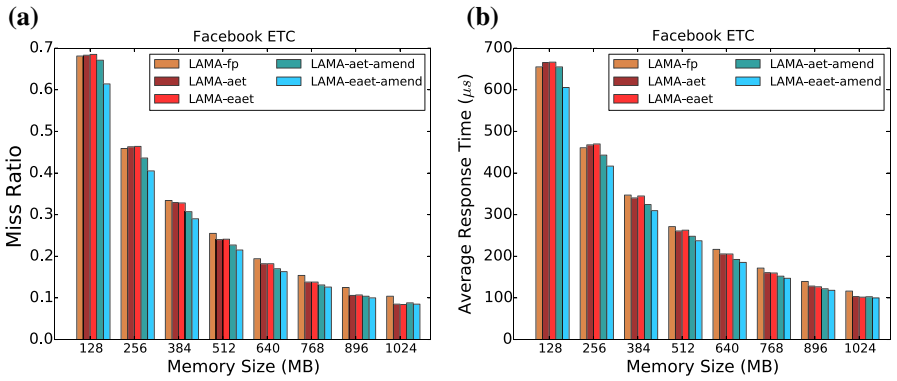

Fig. 6 Stable-state performance under different memory sizes. **a** Miss ratio and **b** average response time

Table 1 Tail latency of LAMA-eaet and LAMA-fp (μs)

Policy	Min	Avg	50th	75th	90th	95th	99th	99.9th	Max
Automove	13	179	19	29	935	988	1137	1348	29,059
LAMA-eaet	14	153	19	19	926	990	1236	1556	29,070
LAMA-fp	15	156	21	23	929	991	1246	1563	46,009[a]

[a]This is an outlier, we simply ignore it

by Memcached-1.4.20, and for simplicity, we will refer to LAMA-eaet-amend as LAMA-eaet.

In Table 1, we list the access latency in the LAMA-eaet, LAMA-fp, and Automove strategies. Compared with Automove, the average latency of LAMA-eaet and LAMA-fp are both lower than it, which is 14.5% and 12.8% lower respectively. Comparing their minimum latency, we can see that Automove is the lowest, this is actually because LAMA-{fp,eaet} need to access the hash table and update the reuse time table, but this latency is actually very low. And we can also see the gap between LAMA-eaet and LAMA-fp's latency of 50–99.9%. The latency of LAMA-fp is always bigger than that of LAMA-eaet because of the larger hash table of LAMA-fp, and it takes significantly more time to update the hash table.

Comparing LAMA-eaet and Automove, LAMA-eaet is dominant on 50–90%, but the Automove's latency is lower at 99% and 99.9%, and the LAMA-fp's situation is similar. The main reason here is because LAMA-{fp,eaet} focuse on reducing the overall miss rate or average response time, which may cause some latency on *cold data* increase.

4.5 Spatial Overhead

We also evaluate the spatial overhead in this part. Table 2 lists the spatial overhead comparisons of LAMA-eaet and LAMA-fp when the working set is 1 GB (LAMA-eaet-amend uses same space as LAMA-eaet). Here LAMA-eaet uses 1% set sampling,

Table 2 Spatial Overhead of LAMA-eaet and LAMA-fp

	LAMA-fp	LAMA-eaet
Items in hash table	2714 K $*$ 24 byte $=$ 62 MB	27 K $*$ 20 byte $=$ 527 KB
Hash table size	$2^{20} * 8$ byte $=$ 8 MB	$2^{16} * 8$ byte $=$ 512 KB
RTH	312 KB	312 KB $*$ 2 $=$ 624 KB
Space for calc MRC	900 KB	512 KB
Total	71 MB	2.1 MB

so the number of objects in the hash table is basically 1% of the LAMA-fp's size, and the size of each object in the hash table is 20 bytes and 24 bytes respectively. The extra 4 bytes in LAMA-fp are because footprint requires additional *first visit time* field.

The final total overhead is 2.1 MB and 71 MB respectively. It can be found that the main spatial overhead of LAMA-fp is the hash table, and 70 MB of the occupied memory is used for the hash table. The cost of this part is actually related to the size of the working set of the access sequence. This is because the footprint algorithm used by LAMA-fp does not support the sampling technique and can't reduce the overhead in this respect. The LAMA-eaet can reduce the spatial overhead by sampling and guarantee the accuracy, the space used only accounts for 3.0% of the LAMA-fp and 0.2% of the total space used.

5 Related Work

Research on cache modeling and MRC construction has focused on LRU and stack algorithms [18,23–26]. Recent approximation algorithms (e.g., CounterStacks [25], SHARDS [26] and AET [18]) make lightweight, continuously-updated MRCs practical for online modeling and control of the LRU caches.

MRC profiling techniques are widely used in different applications. Several studies use on-line MRC analysis for cache partitioning [27,28], page size selection [29], and memory management [30,31]. The memory cache prediction [32] also uses on-line MRC detection for storage workloads. In high-throughput storage systems, fast MRC tracking is always beneficial. One earlier study on optimizing memory allocation is LAMA [15], which uses footprint-based MRC to optimize memory allocation in Memcached, an in-memory key-value store. LAMA shows that MRC-based optimization is superior to the Facebook and Twitter approaches in stable-state performance. Another work called mPart [19] is also an MRC-based memory allocation technique, which uses the AET-based MRCs to optimize memory allocation in a multi-tenant key-value store, and outperforms an existing state-of-the-art sharing model, Memshare [4].

6 Conclusion

We have presented the design and implementation of a lightweight and accurate memory allocation scheme in key-value cache and compare it with a recent solution LAMA.

The evaluation in Memcached shows that we can consume only 3.0% of LAMA space and achieve better performance than LAMA. In Facebook ETC trace, the the average stable-state miss ratio is reduced by 15.0% and the average stable-state response time is reduced by 12.3%.

Acknowledgements We would like to thank anonymous reviewers for their constructive comments and suggestions. The research is supported in part by the National Science Foundation of China (Nos. 61472008, 61672053 and U1611461), Shenzhen Key Research Project No. JCYJ20170412150946024, National Science Foundation CSR1618384 and the National Key R&D Program of China under Grant No. 2018YFB1003505.

References

1. Redis website. https://redis.io/ (2018). Accessed: 2018-07-10
2. Memcached website. http://memcached.org/ (2018). Accessed: 2018-07-10
3. Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., Paleczny, M.: Workload analysis of a large-scale key-value store. In: ACM SIGMETRICS Performance Evaluation Review, vol. 40, pp. 53–64. ACM (2012)
4. Cidon, A., Rushton, D., Rumble, S.M., Stutsman, R.: Memshare: a dynamic multi-tenant key-value cache. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17), pp. 321–334. USENIX Association, Santa Clara (2017). https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon
5. Caching with twemcache. https://blog.twitter.com/engineering/en_us/a/2012/caching-with-twemcache.html (2018). Accessed: 2018-07-10
6. Hart, S., Frachtenberg, E., Berezecki, M.: Predicting Memcached throughput using simulation and modeling. TMS/DEVS'12, pp. 40:1–40:8. http://dl.acm.org/citation.cfm?id=2346616.2346656
7. Saemundsson, T., Bjornsson, H., Chockler, G., Vigfusson, Y.: Dynamic performance profiling of cloud caches. In: SOCC'14, pp. 28:1–28:14. https://doi.org/10.1145/2670979.2671007
8. Jose, J., Subramoni, H., Kandalla, K., Wasi-ur Rahman, M., Wang, H., Narravula, S., Panda, D.K.: Scalable memcached design for InfiniBand clusters using hybrid transports. In: CCGRID'12, pp. 236–243. https://doi.org/10.1109/CCGrid.2012.141
9. Fan, B., Andersen, D.G., Kaminsky, M.: Memc3: Compact and concurrent Memcache with dumber caching and smarter hashing. In: NSDI'13, pp. 371–384. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/fan
10. Lim, K., Meisner, D., Saidi, A.G., Ranganathan, P., Wenisch, T.F.: Thin servers with smart pipes: designing soc accelerators for Memcached. ISCA'13. https://doi.org/10.1145/2485922.2485926
11. Hwang, J., Wood, T.: Adaptive performance-aware distributed memory caching. In: ICAC'13, pp. 33–43. https://www.usenix.org/conference/icac13/technical-sessions/presentation/hwang
12. Zhang, W., Hwang, J., Wood, T., Ramakrishnan, K., Huang, H.: Load balancing of heterogeneous workloads in memcached clusters. In: Feedback Computing'14. https://www.usenix.org/conference/feedbackcomputing14/workshop-program/presentation/zhang
13. Twemcache. https://twitter.com/twemcache (2018). Accessed: 2018-07-10
14. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling Memcache at facebook. In: NSDI'13, pp. 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala
15. Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., Wang, Z.: LAMA: Optimized locality-aware memory allocation for key-value cache. In: USENIX ATC'15, pp. 57–69. https://www.usenix.org/conference/atc15/technical-session/presentation/hu
16. Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., Jiang, S., Wang, Z.: Optimizing locality-aware memory management of key-value caches. IEEE Trans. Comput. **66**(5), 862–875 (2017). https://doi.org/10.1109/TC.2016.2618920
17. Xiang, X., Bao, B., Ding, C., Gao, Y.: Linear-time modeling of program working set in shared cache. In: PACT'11, pp. 350–360

18. Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., Wang, Z.: Kinetic modeling of data eviction in cache. In: USENIX ATC'16
19. Byrne, D., Onder, N., Wang, Z.: mPart: miss-ratio curve guided partitioning in key-value stores. In: Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management, ISMM 2018, pp. 84–95. ACM, New York (2018). https://doi.org/10.1145/3210563.3210571
20. Hu, X., Wang, X., Zhou, L., Luo, Y., Wang, Z., Ding, C., Ye, C.: Fast miss ratio curve modeling for storage cache. ACM Trans. Storage **14**(2), 12:1–12:34 (2018). https://doi.org/10.1145/3185751
21. Pan, C., Hu, X., Zhou, L., Luo, Y., Wang, X., Wang, Z.: PACE: penalty aware cache modeling with enhanced AET. In: APSys'18. ACM (2018). https://doi.org/10.1145/3265723.3265736
22. Mutilate. https://github.com/leverich/mutilate (2018). Accessed: 2018-07-10
23. Mattson, R.L., Gecsei, J., Slutz, D.R., Traiger, I.L.: Evaluation techniques for storage hierarchies. IBM Syst. J. **9**(2), 78–117 (1970). https://doi.org/10.1147/sj.92.0078
24. Niu, Q., Dinan, J., Lu, Q., Sadayappan, P.: Parda: a fast parallel reuse distance analysis algorithm. IPDPS'12, pp. 1284–1294. https://doi.org/10.1109/IPDPS.2012.117
25. Wires, J., Ingram, S., Drudi, Z., Harvey, N.J.A., Warfield, A.: Characterizing storage workloads with counter stacks. In: OSDI'14, pp. 335–349. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires
26. Waldspurger, C.A., Park, N., Garthwaite, A., Ahmad, I.: Efficient MRC construction with SHARDS. In: FAST'15, pp. 95–110
27. Suh, G.E., Devadas, S., Rudolph, L.: Analytical cache models with applications to cache partitioning. In: ICS'01, pp. 1–12
28. Zhang, X., Dwarkadas, S., Shen, K.: Towards practical page coloring-based multicore cache management. In: EuroSys'09, pp. 89–102
29. Cascaval, C., Duesterwald, E., Sweeney, P.F., Wisniewski, R.W.: Multiple page size modeling and optimization. In: PSCT'05, pp. 339–349
30. Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., Kumar, S.: Dynamic tracking of page miss ratio curve for memory management. ACM SIGOPS Oper. Syst. Rev. **38**, 177–188 (2004)
31. Kim, Y.H., Hill, M.D., Wood, D.A.: Implementing stack simulation for highly-associative memories. In: SIGMETRICS'91, pp. 212–213
32. Bjornsson, H., Chockler, G., Saemundsson, T., Vigfusson, Y.: Dynamic performance profiling of cloud caches. In: SOCC'13