Optimizing Locality-aware Memory Management of Key-value Caches

Xiameng Hu, *Student Member, IEEE,* Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, *Member, IEEE,* Song Jiang, Zhenlin Wang

Abstract—The in-memory cache system is a performance-critical layer in today's web server architecture. Memcached is one of the most effective, representative, and prevalent among such systems. An important problem is on its memory allocation. The default design does not make the best use of the memory. It is unable to adapt when the demand changes, a problem known as *slab calcification*.
This paper introduces locality-aware memory allocation (LAMA), which addresses the problem by first analyzing locality of Memcached's requests and then reassigning slabs to minimize the miss ratio or the average response time. By evaluating LAMA using various industry and academic workloads, the paper shows that LAMA outperforms existing techniques in the steady-state performance, the speed of convergence, and the ability to adapt to request pattern changes, and overcome slab calcification. The new solution is close to optimal, achieving over 98% of the theoretical potential. Furthermore, LAMA can also be adopted in resource partition to guarantee quality-of-service (QoS).

Index Terms—Data Locality, Memory Allocation, Key-value Cache, Quality of Service.

1 Introduction

In today's web server architecture, distributed in-memory caches are vital components to ensure low-latency service for user requests. Many companies use in-memory caches to support web applications. For example, the time to retrieve a web page from a remote server can be reduced by caching the web page in server's memory since accessing data in memory cache is much faster than querying a back-end database. Through this cache layer, the database query latency can be reduced as long as the cache is sufficiently large to maintain a high hit rate.

Memcached [1] is a commonly used distributed in-memory key-value cache system, which has been deployed in Facebook, Twitter, Wikipedia, Flickr, and many other internet companies. It has been proposed to use Memcached as an additional layer to accelerate systems such as Hadoop, MapReduce, and even virtual machines [2], [3], [4]. Memcached splits the memory cache space into different *classes* to store variable-sized objects as *items*. Initially, each class obtains its own memory space by requesting free *slabs*, 1MB each, from the allocator. Each allocated slab is divided into *slots* of equal size. According to the slot size, the slabs are categorized into different classes, from Class 1 to Class n, where the slot size increases exponentially. A newly incoming item is admitted into a class whose slot size is the best fit of the item size. If there is no free space in the class, a currently cached item has to be first *evicted* from the class of slabs following the LRU policy. In this design, the number of slabs in each class represents the memory space that has been allocated to it.

- *X. Hu, X. Wang, L. Zhou and Y. Luo are with Peking University, Beijing, China.*
  E-mail: {hxm,wxl,lanzhou,lyw}@pku.edu.cn
- *C. Ding is with University of Rochester, Rochester, NY.*
  E-mail: cding@cs.rochester.edu
- *S. Jiang is with Wayne State University, Detroit, MI.*
  E-mail: sjiang@wayne.edu
- *Z. Wang is with Michigan Technological University, Houghton, MI.*
  E-mail: zlwang@mtu.edu

As memory is much more expensive than external storage devices, the system operators need to maximize the efficiency of memory cache. They need to know how much cache space should be deployed to meet the service-level-agreements (SLAs). Default Memcached fills the cache at the cold start based on the demand. We observe that this demand-driven slab allocation does not deliver optimal performance, which will be explained in Section 2.1. Performance prediction [5], [6] and optimization [7], [8], [9], [10], [11] for Memcached have drawn much attention recently. Some studies focus on profiling and modeling the performance under different cache capacities [6]. In the presence of workload changing, default Memcached server may suffer from a problem called *slab calcification* [12], in which the slab allocation cannot be adjusted to fit the change of access pattern as the old slab allocation may not work well for the new workload. To avoid the performance drop, 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], [15]. We will analyze several state-of-the-art solutions in Section 2. We find that these approaches are still far behind a theoretical optimum as they do not exploit the locality inherent in the Memcached requests.

We propose a novel, dynamic slab allocation scheme, *locality-aware memory allocation* (LAMA), based on a recent advance on measurement of data locality [16] described in Section 2.2. This study provides a low-overhead yet accurate method to model data locality and generate miss ratio curves (MRCs). Miss ratio curve (MRC) reveals relationship between cache sizes and cache miss ratios. With MRCs for all classes, the overall Memcached performance can be modeled in terms of different class space allocations, and it can be optimized by adjusting individual classes' allocation. We have developed a prototype system based on Memcached-1.4.20 with the locality-aware allocation of memory space. Further more, we propose to use the locality analysis framework in LAMA to guide the server resource partition for latency sensitive applications. The experimental results show that LAMA can achieve over 98% of the theoretical potential and the server partition method is quite efficient for QoS guarantee.

2 Background

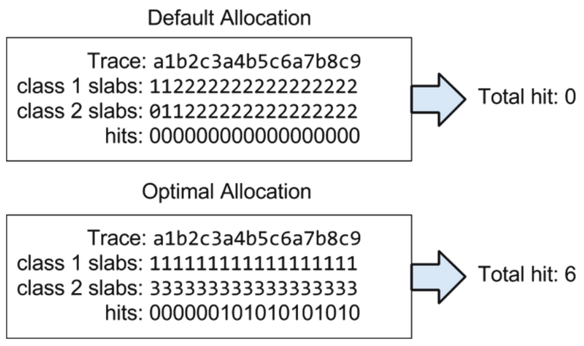
This section summarizes the Memcached's allocation design and its recent optimizations, which we will compare against LAMA, and a locality theory, which we will use in LAMA.

2.1 Memory Allocation in Memcached

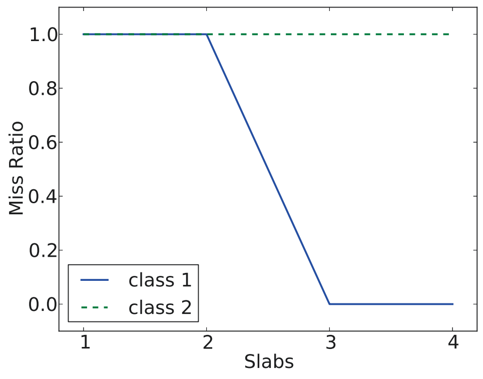
Default Design In most cases, Memcached is demand filled. The default slab allocation is based on the number of items arriving in different classes during the cold start period. However, we note that in real world workloads, a small portion of the items appears in most of the requests. For example, in the Facebook ETC workload [17], 50% of the items occur in only 1% of all requests. It is likely that a large portion of real world workloads have similar data locality. The naive allocation of Memcached may lead to low cache utilization due to negligence of data locality in its design. Figure 1 shows an example to illustrate the issue of a naive allocation. Let us assume that there are two classes of slabs to receive a sequence of requests. In the example, the sequence of items for writing into Class 1 is "*123456789...*", and the sequence into Class 2 is "*abcabcabc...*". We also assume that each slab holds only one item in both classes for the sake of simplicity, and there are a total of four slabs. If the access rates of the two classes are the same, the combined access pattern would be "*a1b2c3a4b5c6a7b8c9...*". In the default allocation, every class will obtain two slabs (items) because they both store two objects during the cold start period. Note that the reuse distance of any request is larger than two for both classes. The number of hits under naive allocation would be 0. As the working set size of Class 2 is 3, the hit ratio of Class 2 will be 100% with an allocation of 3 slabs according to the MRC in Figure 1(b). If we reallocate one slab from Class 1 to Class 2, the working set of Class 2 can be fully cached and every reference to Class 2 will be a hit. Although the hit ratio of Class 1 is still 0%, the overall hit ratio of cache server will be 50%. This is much higher than the hit ratio of the default allocation which is 0%. This example motivates us to allocate space to the classes of slabs according to their data locality.

Automove The open-source community has implemented an automatic memory reassignment algorithm (Automove) in a former version Memcached [18]. In every 10 seconds window, the Memcached server counts the number of evictions in each class. If a class takes the highest number of evictions in three consecutive monitoring windows, a new slab is reassigned to it. The new slab is taken from the class that has no evictions in the last three monitoring stages. This policy is greedy but lazy. In real workloads, it is hard to find a class with no evictions for 30 seconds. Accordingly, the probability for a slab to be reassigned is extremely low.

Since Memcached-1.4.25, the slab automover has gotten a very large update. A background process periodically reclaims the space held by expired data items and produces free slabs out of the reclaimed space for future reassignment. The new automover behaves like a garbage collector. This new feature increases the amount of free space, and it is orthogonal to reallocation of the existing space, which is the problem we address.


Default Allocation
Trace: a1b2c3a4b5c6a7b8c9
class 1 slabs: 112222222222222222
class 2 slabs: 011222222222222222
hits: 000000000000000000
Total hit: 0
Optimal Allocation
Trace: a1b2c3a4b5c6a7b8c9
class 1 slabs: 111111111111111111
class 2 slabs: 333333333333333333
hits: 000000101010101010
Total hit: 6

(a) Access detail for different allocation



(b) MRCs for Class 1&2

Fig. 1: Drawbacks of default allocation

Twitter Policy To tackle the slab calcification problem, Twitter's implementation of Memcached (Twemcache) [13] introduces a new eviction strategy to avoid frequently restarting the server. Every time a new item needs to be inserted but there is no free slabs or expired ones, a random slab is selected from all allocated slabs and reassigned to the class that fits the new item. This random eviction strategy aims to balance the eviction rates among all classes to prevent performance degradation due to workload change. The operator no longer needs to worry about reconfiguring the cache server when calcification happens. However, random eviction is aggressive since frequent slab evictions can cause performance fluctuations, as observed in our experiments in Section 5. In addition, a randomly chosen slab may contain data that would have been future hits. The random reallocation apparently does not consider the locality.

Periodic Slab Allocation (PSA) Carra et al. [14] address some disadvantages of Twemcache and Automove by proposing *periodic slab allocation* (PSA). At any time window, the number of requests of Class i is denoted as R_i and the number of slabs allocated to it is denoted as S_i. The risk of moving one slab away from Class i is denoted as R_i/S_i. Every M misses, PSA moves one slab from the class with the lowest risk to the class with the largest number of misses. PSA has an advantage over Twemcache and Automove by picking the most promising candidate classes to reassign slabs. It aims to find a slab whose reassignment to another class dose not result in more misses. Compared with Twemcache's random selection strategy, PSA chooses the lowest risk class to minimize the penalty. However, PSA has a critical drawback: classes with the highest miss rates can also be the ones with the lowest risks. In this case, slab reassignment will only occur between these classes. Other classes will stay untouched and unoptimized since there is no chance to adjust slab allocation among them. Figure 2 illustrates a simple example where PSA can get stuck. Assume that a cache server consists of three slabs and every slab contains only one item. The global access trace is "$(aa1aa2baa1aa2aa1ba2)^*$", which is composed of Class 1 "*121212...*" and Class 2 "$(aaaabaaaaaaba)^*$". If Class 1 has taken only one slab (item) and Class 2 has taken two items, Class 1 would have the highest miss rate and the lowest risk. The system will be in a state with no slab reassignment. The overall system hit ratio under this allocation will be 68%. However, if a slab (item) were to be reassigned from Class 2 to Class 1, the hit ratio will increase to 79% since the working set size of Class 1 is 2. Apart from this weak point, in our experiments, PSA shows good adaptability for slab calcification since it can react quickly to workload changing. However, since the PSA algorithm lacks a global perspective for slab assignment, the performance still falls short when compared with our locality-aware scheme.

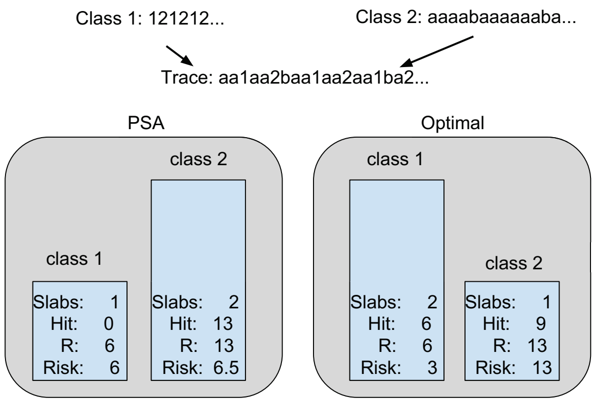


Fig. 2: Drawbacks of PSA

Facebook Policy

Facebook's optimization of Memcached [15] uses adaptive slab allocation strategy to balance item age. In their design, if a class is currently evicting items, and the next item to be evicted was used at least 20% more recently than the average least recently used item of all other classes, this class is identified as needing more memory. The slab holding the overall least recently used item will be reassigned to the needy class. This algorithm balances the age of the least recently used items among all classes. Effectively, the policy approximates the global LRU policy, which is inherently weaker than optimal as shown by Brock et al. using the footprint theory we will describe next [19].

The policies of default Memcached, Twemcache, Automove, and PSA all aim to equalize the eviction rate among size classes. The Facebook policy aims to equalize the age of the oldest item in size classes. We call the former performance balancing and the latter age balancing. Later in the evaluation section, we will compare these policies and show their relative strengths and weaknesses.

2.2 The Footprint Theory

The locality theory is by Xiang et al., who define a metric called *footprint* and propose a linear time algorithm to measure it [16] and a formula to convert it into the miss ratio [20]. Next we give the definition of footprint and show its use in predicting the miss ratio.

The purpose of the footprint is to quantify the locality in a period of program execution. An execution trace is a sequence of memory accesses, each of which is represented by a memory address. Accesses can be tagged with logical or physical time. The logical time counts the number of accesses from the start of the trace. The physical time counts the elapsed time. An execution window is a sub-sequence of consecutive accesses in the trace.

The locality of an execution window is measured by the working-set size , which is the amount of data accessed by all its accesses [21]. The footprint is a function $fp(w)$ as the average working-set size for all windows of the same length w. While different window may have different working-set size, $fp(w)$ is unique. It is the expected working-set size for a randomly selected window.

Consider a trace abcca. Each element is a window of length $w = 1$. The working-set size is always 1, so $fp(1) = 5/5 = 1$. There are 4 windows of length $w = 2$. Their working-set sizes are 2, 2, 1, and 2. The average, i.e., the footprint, is $fp(2) = 7/4$. For greater window lengths, we have $fp(3) = 7/3$ and $fp(w) = 3$ for $w = 4, 5$, where 5 is the largest window length, i.e., the length of the trace. We also define $fp(0) = 0$.

Although the footprint theory is proposed to model locality of data accesses of a program, the same theory can be applied in modeling the locality of Memcached requests where data access addresses are replaced by the keys. The linear time footprint analysis leads to linear time MRC construction and thus a low-cost slab allocation prediction, as discussed next.

3 Locality-aware Memory Allocation

This section describes the design details of LAMA.

3.1 Locality-based Caching

Memcached allocates the memory at the granularity of a slab, which is 1MB in the default configuration. For every size class, Memcached allocates its items in its collection of slabs. The items are ordered in a priority list based on their last access time, forming an LRU chain. The head item of the chain has the most recent access, and the tail item the least recent access. When all the allocated slabs are filled, eviction will happen when a new item is accessed, i.e. a cache miss. When the tail item is evicted, its memory is used to store the new item, and the new item is re-inserted at the first position to become the new head.

In a web-service application, some portion of items may be frequently requested. Because of their frequent access, the hot items will reside near the top of the LRU chain and hence be given higher priority to cache. A class' capacity, however, is important, since hot items can still be evicted if the amount of allocated memory is not large enough.

A slab may be reassigned from one size class to another. The $SlabReassign$ routine in Memcached releases a slab used in a size class and gives it to another size class. The reassignment evicts all the items that are stored in the slab and removes these items from the LRU chain. The slab is now unoccupied and changes hands to store items for the new size class.

Memcached may serve multiple applications at the same time. The memory is shared. Since requests are pooled, the LRU chain gives the priority of all items based on the aggregate access from all programs.

3.2 MRC Profiling

We split the global access trace into different sub-traces according to their classes. With the sub-trace of each class, we generate the MRCs as follows. We use a hash table to record the last access time of each item. With this hash table, we can easily compute the reuse time distribution r_t, which represents the number of accesses with a reuse time t. 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 [16]:

$$\begin{aligned} \mathit{fp}(w) &= m - \frac{1}{n-w+1}(\sum_{i=1}^{m}(f_i - w)I(f_i > w) \\ &+ \sum_{i=1}^{m}(l_i - w)I(l_i > w) \\ &+ \sum_{t=w+1}^{n-1}(t-w)r_t) \end{aligned} \tag{1}$$

The symbols are defined as:

- f_i: the first access time of the i-th datum
- l_i: the *reverse* last access time of the i-th datum. If the last access is at position x, $l_i = n + 1 - x$, that is, the first access time in the reverse trace.
- $I(p)$: the predicate function equals to 1 if p is true; otherwise 0.
- r_t: the the number of accesses with a reuse time t.

Now we can profile the MRC using *fp* distribution. The miss ratio for cache size of x is the fraction of reuses that have an average footprint larger than x:

$$MRC(x) = 1 - \frac{\sum_{\{t|fp(t)<x\}} r_t}{n} \tag{2}$$

For example, we show the footprint-based MRC calculation for cyclic reference pattern "$(abc)^*$" in Table 1. This is the trace of Class 2 in Figure 1. If the trace length is infinite, the miss ratio is 100% if the footprint is greater than the cache size, but 0% otherwise.

TABLE 1: An example of MRC calculation by footprint

x	0	1	2	3	> 3
fp(x)	0	1	2	3	3
MRC(x)	1	1	1	0	0

3.3 Target Performance

We consider two types of target performance: the total miss ratio and the average response time.

If Class i has taken S_i slabs, and I_i represents the number of items per slab in Class i. Then there should be $S_i * I_i$ items in this class. The miss ratio of this class should be $MR_i = MRC_i(S_i * I_i)$. Let the number of requests of Class i be R_i. The total miss ratio is calculated as:

$$Miss\ Ratio = \frac{\sum_{i=1}^{n} R_i * MR_i}{\sum_{i=1}^{n} R_i} = \frac{\sum_{i=1}^{n} R_i * MRC_i(S_i * I_i)}{\sum_{i=1}^{n} R_i} \tag{3}$$

Let the average request hit time for Class i be $T_h(i)$, and the average request miss time (including retrieving data from database and setting back to Memcached) be $T_m(i)$. The average request time ART_i of Class i now can be presented as:

$$ART_i = MR_i * T_m(i) + (1 - MR_i) * T_h(i) \tag{4}$$

The overall ART of the Memcached server is:

$$ART = \frac{\sum_{i=1}^{n} R_i * ART_i}{\sum_{i=1}^{n} R_i} \tag{5}$$

We target the overall performance by all size classes rather than equal performance in each class. The metrics take into account the relative total demands for different size classes. If we consider a typical request as the one that has the same proportional usage, then the optimal performance overall implies the optimal performance for a typical request.

3.4 Optimal Memory Reallocation

When a Memcached server is started, the available memory is allocated by demand. Once the memory is fully allocated, we have a partition among all size classes. LAMA periodically measures the MRCs and reallocating the memory.

The optimization problem is as follows. Given the MRC for each size class, how to divide the memory among all size classes so that the target performance is maximized, i.e., the total miss ratio or the average response time is minimized?

The reallocation algorithm has two steps:

Step 1: Cost Calculation First we split the access trace into sub-traces based on their classes. For each sub-trace $T[i]$ of Class i, we use the procedure described in Section 3.2 to calculate the miss ratio $MR[i][j]$ when allocated j slabs, $0 \leq j \leq$ *MAX*, where *MAX* is the total number of slabs. We compute the cost for different optimization targets.

To minimize total misses, $Cost[i][j]$ is the number of misses for Class i given its allocation j as follows:

$$Cost[i][j] \leftarrow MR[i][j] * length(T[i]).$$

To minimize ART, $Cost[i][j]$ is the average access time of Class i as follows:

$$\begin{aligned} Cost[i][j] \leftarrow (&MR[i][j] * T_m[i] + \\ &(1 - MR[i][j]) * T_h[i]) * length(T[i]) \end{aligned}$$

Algorithm 1 Locality-aware Memory Allocation

```
Input: $Cost[][]$ // cost function, could be $OPT\_MISS$ or $OPT\_ART$
Input: $MAX$ // total number of slabs
Ensure: $Slabs_{new}[]$ // optimal slabs allocation
1: function OptimalAllocation($Cost[][], S_{old}[], MAX$)
2:     $F[][] \leftarrow +\infty$
3:                         $\triangleright$ $F[][]$ minimal cost for Class $1..i$ using $j$ slabs
4:     for $i \leftarrow 1..n$ do
5:         for $j \leftarrow 1..MAX$ do
6:             for $k \leftarrow 0..j$ do
7:                 $Temp \leftarrow F[i-1][j-k] + Cost[i][k]$
8:                                         $\triangleright$ Give $k$ slabs to Class $i$.
9:                 if $Temp < F[i][j]$ then
10:                    $F[i][j] \leftarrow Temp$
11:                    $B[i][j] \leftarrow k$
12:                                     $\triangleright$ $B[][]$ saves the slab allocation.
13:                end if
14:            end for
15:        end for
16:    end for
17:    $Temp \leftarrow MAX$
18:    for $i \leftarrow n..1$ do
19:        $Slabs_{new}[i] \leftarrow B[i][Temp]$
20:        $Temp \leftarrow Temp - B[i][Temp]$
21:    end for
22:    return $Slabs_{new}[]$
23: end function
```

Step 2: Reallocation We design a dynamic programming algorithm to find new memory allocation (Algorithm 1). Lines 4 to 16 show a triple nested loop. The outermost loop iterates the set of size classes i from 1 to n. The middle loop iterates the number of slabs j from 1 to *MAX*. The target function, $F[i][j]$, stores the optimal cost of allocating j slabs to i size classes. The innermost loop iterates the allocation for the latest size class to find this optimal value.

Once the new allocation is determined, it is compared with the previous allocation to see if the performance improvement is above a certain threshold. If it is, slabs are reassigned to change the allocation. Through this procedure, LAMA reorganizes multiple slabs across all size classes. The dynamic programming algorithm is similar to Brock et al. [19] but for a different purpose.

In real deployment of the algorithm, we can set N as an upper bound on the number of reassignments for each repartitioning and each reassignment is limited to 1 slab (In Section 5.4, we will discuss a reassignment granularity of r slabs). This will avert the cost of reassigning too many slabs at one time. At each reallocation, we choose N slabs with the lowest risk. We use the risk definition of PSA, which is the ratio between reference rate and number of slabs for each class. The reallocation is global, since multiple candidate slabs are selected from possibly many size classes. In contrast, PSA selects a single candidate from one size class. With the reassignment constraint, the performance drop due to massive eviction of useful data is avoid. This design also narrows the search space of possible solutions in Algorithm 1. As we can infer from the triple nested loop, the time complexity of the optimization is $O(n * MAX^2)$, where n is the number of size classes and *MAX* is the total number of slabs. With the reassigning upper bound N, the inner loop from line 6 to 14 do not have to search the whole memory space, but the space with a radius of N slabs from current allocation. Therefore, the time complexity of Algorithm 1 can be improved to $O(n * MAX * N)$, which is efficient in large scale system.

The bound N is the maximal number of reassignments. In the steady state, the reallocation algorithm may decide that the current allocation is the best possible and does not reassign any slab. The number of actual reassignments can be 0 or any number not exceeding N.

Algorithm 1 optimizes the overall performance. The solution may not be fair, i.e., different miss ratios across size classes. Fairness is not a concern at the level of memory allocation. Facebook solves the problem at a higher level by running a dedicated Memcached server for critical applications [17]. We introduce locality-aware server partition to guarantee the QoS of a critical application in Section 4. If fairness is a concern, Algorithm 1 can use a revised cost function to discard unfair solutions and optimize both for performance and fairness. A recent solution is the baseline optimization by Brock et al. [19] and Ye et al. [22].

3.5 Performance Prediction

Despite the optimal allocation, the locality analysis framework in LAMA can also be adopted to predict the performance of the default Memcached under different memory space. This can avoid frequently restarting sever for performance testing. Using Equation 1 in Section 3.2, the average footprint of any window size can be obtained. For a stable access pattern, we define the request ratio of Class i as q_i. Let the number of requests during the cold start period be M. The allocation for Class i by the default Memcached is the number of items it requests during this period. We predict this allocation as $fp_i(M * q_i)$. The length M of the cold-start period, i.e., the period during which the memory is completely allocated, satisfies the following equation:

$$\sum_{i=1}^{n} fp_i(M * q_i) = C \tag{6}$$

Once we get the expected items (slabs) each class can take, the system performance can be predicted by Equation 3. By predicting M and the memory allocation for each class, we can predict the performance of default Memcached for all memory sizes. The predicted allocation is similar to the natural partition of CPU cache memory, as studied in [19]. Using the footprint theory, our approach delivers high accuracy and low overhead. This is important for a system operator to determine how many caches should be deployed to achieve required Quality of Service (QoS).

4 Locality-aware Server Partition

A typical Memcached cluster can be shared by multiple applications or web services of different data access patterns. Each of them may have different quality-of-service (QoS) requirements. In each Memcached server, its cached data blocks come from different applications sharing the server. They are all managed by the LRU eviction policy in individual size classes. In this scenario, applications in the same Memcached server compete for the same memory. The memory occupation of each application is determined by factors including access frequency or reuse distance distribution. As observed, Facebook applications negatively interfere with each other [15]. For example, an application with a high

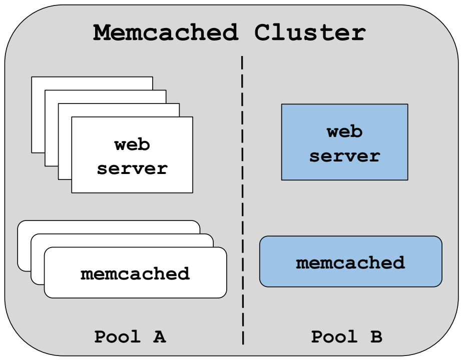


Fig. 3: Inter-Server Partition

access frequency may occupy more memory than an application with a low access frequency. Because of this sharing, one may not be able to guarantee the QoS of performance-critical applications. Among web services, some applications are latency sensitive and require a fast user response. For these applications, a high hit ratio in the cache layer is necessary to guarantee QoS. They should be prevented from sharing a Memcached server with unknown applications. Instead, they should be allocated dedicated memory.

To guarantee the QoS of performance-critical applications, one solution is to partition Memcached servers into separate pools and dedicate some of the pools for certain applications. Absent of sharing, these applications are not affected by interference from dynamic memory sharing. This design has been adopted by Facebook to protect the valuable keys by a memory pool [15]. Figure 3 illustrates the layout of this *inter-server partition* scheme.

For smaller scale deployment of Memcached, a finer grain way to do the isolation is partitioning the memory space within each server. Instead of assigning servers among applications, all Memcached servers serve all applications. The partition happens in the memory of each server, as shown in Figure 4. The difference between sharing a server and partitioning a server is similar to CPU cache partition and sharing (for a recent study, see [19]). Every slab in a Memcached server is assigned to a specific application (or a group of applications). In a partitioned server, an application can have its exclusive LRU chain, which is constructed for the slabs allocated to it. This design is similar to multiple Memcached instances running on a single physical server, each instance only manages its own memory for one or several applications.

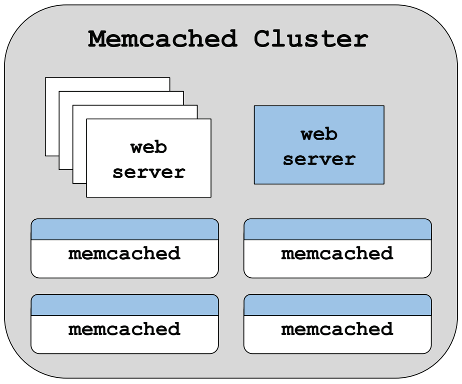


Fig. 4: Intra-Server Partition

The intra-server partition provides isolation for different applications in a single server. Although they are still "sharing" the Memcached server, partition protects the valuable data in a performance-critical application from being evicted by other applications. This design also provides the ability of dynamically adjusting the partition to respond to demand changes.

Although an improved eviction policy, CAMP [23], is proposed to take size and cost into consideration to prevent the eviction of valuable data from performance-critical applications, the design motivation of CAMP is to avoid human involvement to partition the available memory into disjoint pools. It shows that heuristic memory partition is suboptimal compare to well designed eviction policy. But this is a less predictable strategy. It is hard to predict and control the hit ratios of several applications when they are running and sharing the same memory space. However, with our performance prediction technique, we will show how QoS predictable memory partition can be done dynamically.

Algorithm 2 Performance Guarantee Partition

```
Input: $MR_{req}$ // Miss ratio requirement for QoS guarantee
Input: $UP$ // The upper bound of slabs occupation
Ensure: $Slabs_{req}[]$ // Number of slabs required
 1: function ServerPartition($MR_{req}, UP$)
 2:     $Left \leftarrow 0$
 3:     $Right \leftarrow UP$
 4:     $Slabs_{req}[] \leftarrow OptimalAllocation(Cost[][], UP)$
 5:     $MR_{new} \leftarrow 0$
 6:     for $i \leftarrow n..1$ do
 7:         $MR_{new} \leftarrow MR_{new} + MR[i][Slabs_{req}[i]]$
 8:     end for
 9:     if $MR_{new} > MR_{req}$ then
10:         return $MORE\_SPACE\_REQUIRED$
11:     end if
12:     while $Left + 1 < Right$ do
13:         $Mid \leftarrow (Left + Right)/2$
14:         $Slabs_{req}[] \leftarrow OptimalAllocation(Cost[][], Mid)$
15:         $MR_{new} \leftarrow 0$
16:         for $i \leftarrow n..1$ do
17:             $MR_{new} \leftarrow MR_{new} + MR[i][Slabs_{req}[i]]$
18:         end for
19:         if $MR_{new} > MR_{req}$ then
20:             $Left \leftarrow Mid$
21:         else
22:             $Right \leftarrow Mid$
23:         end if
24:     end while
25:     $Slabs_{req}[] \leftarrow OptimalAllocation(Cost[][], Right)$
26:     return $Slabs_{req}[]$
27: end function
```

We use intra-server partition to isolate the latency-sensitive application in order to satisfy its QoS requirement. Since the intra-server partition does not change the key-to-server mapping, the partition is determined by each individual Memcached server. There is no need for a global scheduler to gather MRC information from all applications. During the execution, each sever keeps tracking the access pattern to make sure that the current partition is effective and efficient. The optimal memory allocation we described in Section 3.4 is used here to partition the memory among size classes within an application. The intra-server partition is an upper-level partition that partitions the memory for performance-critical applications. The memory is partitioned at two levels first for QoS among applications and then for optimal allocation among size classes. Both are based on the same locality analysis. Algorithm 2 shows the intra-server partition algorithm. It uses binary search to find the least number of slabs required for a QoS target. When given the number of slabs, Algorithm 1 is invoked to conduct the optimal slab allocation for the minimal miss ratio. A higher or lower miss ratio requires searching into more space or less space, respectively. If the required slab partition ($Slabs_{req}$) is smaller than current partition, the application will release the extra slabs. Otherwise, the system will assign new slabs to it. To prevent any application from taking all the slabs and starving other applications, an upper bound of slab occupation for each application may be set. If there are multiple applications that are critical and require QoS at the same time, and the Memcached servers cannot provide enough memory to satisfy all of their requirements, a priority-based scheduling may be used.

5 Evaluation

In this section, we evaluate LAMA in detail, including describing the experimental setup for evaluation and comprehensive evaluation results and analysis.

5.1 Experimental setup

LAMA Implementation We have implemented LAMA in Memcached-1.4.20 (Section 5.2 to 5.7). To evaulate LAMA latencies and compare it with the most recent Automove design, we have also implemented LAMA in a more recent version of Memcached-1.4.29 (Section 5.8), which adopts a garbage collector to recollect expired items. For MRC analysis, we used to record the access trace using a circular buffer. This is space consuming and requires a mutex lock for each thread to update the buffer. We replace this design by a lock free hash table to track the first reference and last reference as well as the reuse time for each item. The other statistics to calculate footprint are updated by every access in a lock free manner too. This avoids recording the trace and the mutex lock for every thread. Beside, the MRC calculation is much faster from those updated data structures than a buffered trace. In our test, when we need MRCs for repartition, all the MRCs can be calculated with in a second. The space cost of these data structures depends on the size of the items being analyzed. It is 1% - 2% of all memory depending for the workload we use. The MRCs calculation and slab reassignment (Algorithm 1) are performed in the slab rebalancer thread of Memcached. The overhead is negligible, both in time and in space.

System Setup To evaluate the efficiency of LAMA and other strategies, we use a single node, Intel(R) Core(TM) I7-3770 machine with 4 cores, 3.4GHz, 8MB shared LLC with 16GB memory. To evaluate the scalability of LAMA, we use a Dell PowerEdge R720 with ten-core 2.50GHz Intel Xeon E5-2670 v2 processors and 256 GB of RAM.

For small scale test, we measure both the miss ratio and the response time, as defined in Section 3.4. In order to measure the latter, we set up a database as the backing store to the Memcached server. The response time is the wall-clock time used for each client request by the server, including the cost of the database access. For scalability test, we measure the maximum throughput of Memcached with LAMA, as well as the 95% tail latency to verify the overhead of LAMA in a large scale system. In our evaluation, all the Memcached instances are running on local ports with 4 server threads and the database is running from another server on the local network.

Workloads Three workloads are used for different aspects of the evaluation:

- **The *Facebook ETC* workload to test the steady-state performance.** It is generated using Mutilate [24], which emulates the characteristics of the ETC workload at Facebook. ETC is the closest workload to a general-purpose one, with the highest miss ratio in all Facebook's Memcached pools. It is reported that the installation at Facebook uses hundreds of nodes in one cluster [17]. We set the workload to have 50 million requests to 7 million data objects.
- **A *three-phase* workload to test dynamic allocation.** It is constructed based on Carra et al. [14]. It has 200 million requests to data items in two working sets, each of which has 7 million items. The first phase only accesses the first set following a generalized Pareto distribution with location $\theta = 0$, scale $\phi = 214.476$ and shape $k = 0.348238$, based on the numbers reported by Atikoglu et al. [17]. The third phase only accesses the second set following the Pareto distribution $\theta = 0$, ϕ = 312.6175 and $k = 0.05$. The middle, transition phase increasingly accesses data objects from the second set.
- **A *large-scale-test* workload to test scalability.** We use the Data Caching Benchmark of CloudSuite [25]. It simulates the behavior of a Twitter caching server using the twitter dataset. The original dataset consumes 300MB of server memory, while we scale up the dataset to 150GB (scaling factor of 500). We set up the benchmark loader with 200 TPC/IP connections, and a get/set ration of 0.8.

5.2 Facebook ETC Performance

We test and compare LAMA with the policies of default Memcached, Automove, PSA, Facebook, and Twitter's Twemcache (described in Section 2). In our experiments, Automove finds no chance of slab reassignment, so it has the same performance as Memcached. LAMA has two variants: LAMA_OPT_MR, which tries to minimize the miss ratio; and LAMA_OPT_ART, which tries to minimize the average response time. Figures 5 and 6 show the miss ratio and ART over time from the cold-start to steady-state performance. The total memory is 512MB.

The default Memcached and PSA are designed to balance the miss ratio among size classes. LAMA tries to minimize the total miss ratio. Performance optimization by LAMA shows a large advantage over performance balancing by Memcached and PSA. If we compare the steady-state miss ratio, LAMA_OPT_MR is 47.20% and 18.08% lower than Memcached and PSA. If we compare the steady-state ART, LAMA_OPT_ART is 33.45% and 13.17% lower.

There is a warm-up time before reaching the steady state. LAMA repartitions at around every 300 seconds and reassigns up to 50 slabs. We run PSA at 50 times the LAMA frequency, since PSA reassigns 1 slab each time. LAMA, PSA and Memcached converge to the steady state at the same speed. Our implementation of optimal allocation (Section 4.6) shows that this speed is the fastest.

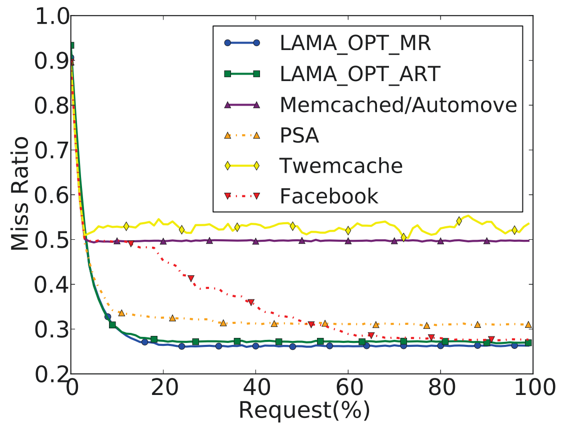


Fig. 5: Miss ratio from cold-start to steady state

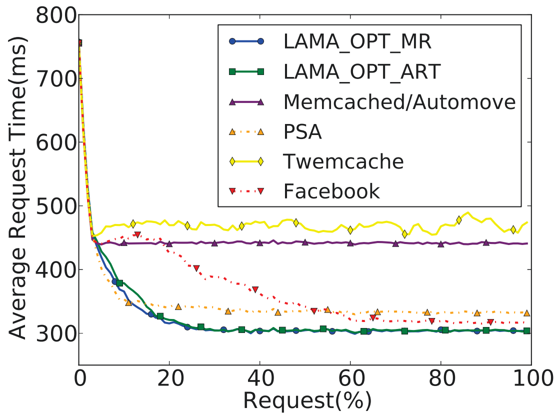


Fig. 6: Average response time from cold-start to steady state

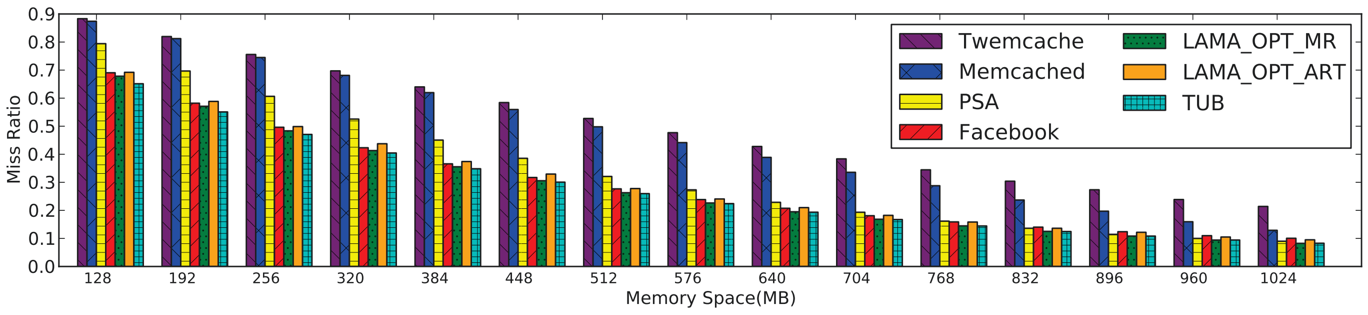


Fig. 7: Steady-state miss ratio with different memory sizes

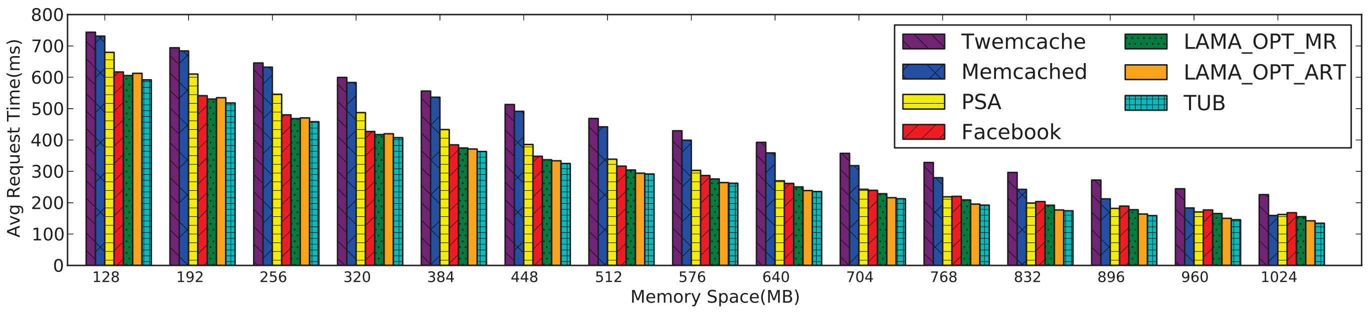


Fig. 8: Steady-state average response time when using different amounts of memory

The Facebook method differs from others in that it seeks to equalize the age of the oldest items in each size class. In the steady state, it performs closest to LAMA, 5.4% higher than LAMA_OPT_MR in the miss ratio and 6.7% higher than LAMA_OPT_ART in the average response time. The greater weakness, however, is the speed of convergence, which is about 4 times slower than LAMA and the other methods.

Twemcache uses random rather than LRU replacement. In this test, the performance does not stabilize as well as the other methods, and it is generally worse than the other methods. Random replacement can avoid slab calcification, which we consider in Section 5.5.

Next we compare the steady-state performance for memory sizes from 128MB to 1024MB in 64MB increments. Figures 7 and 8 show that the two LAMA solutions are consistently the best at all memory sizes. The margin narrows in the average response time when the memory size is large. Compared with Memcached, LAMA reduces the average miss ratio by 41.9% (22.4%–46.6%) for the same cache size, while PSA and Facebook reduce the miss ratio by 31.7% (9.1%–43.9%) and 37.6% (21.0%–47.1%). For the same or lower miss ratio, LAMA saves 40.8% (22.7%–66.4%) memory space, PSA and Facebook save 29.7% (14.6%–46.4%) and 36.9% (15.4%–55.4%) respectively.

Heuristic solutions show strength in specific cases. Facebook improves significantly over PSA for smaller memory sizes (in the steadstate). With 832MB and larger memory, PSA catches up and slightly outperforms Facebook. At 1024MB, Memcached has a slightly faster ART than both PSA and Facebook. The strength of optimization is universal. LAMA maintains a clear lead against all other methods at all memory sizes.

Compared to previous methods on different memory sizes, LAMA converges among the fastest and reaches a greater steady-state performance. The steady-state graphs also show the theoretical upper bound performance (TUB), which we discuss in Section 5.6.

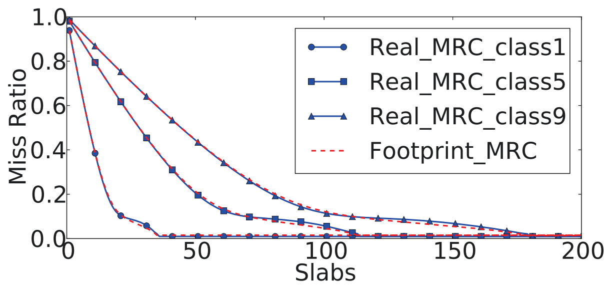


Fig. 9: MRCs for class 1&5&9

TABLE 2: prediction miss ratio vs. real miss ratio

Capacity	Real	Prediction	Accuracy
128MB	87.56%	88.21%	99.26%
256MB	74.68%	75.40%	99.05%
384MB	62.34%	62.63%	99.54%
512MB	50.34%	50.83%	99.04%
640MB	39.36%	39.52%	99.60%
768MB	29.04%	29.27%	99.21%
896MB	20.18%	20.61%	97.91%
1024MB	13.36%	13.46%	99.26%

5.3 MRC Accuracy

To be optimal, LAMA must have the accurate MRC. We compare the LAMA MRC, obtained by sampling and footprint version, with the actual MRC, obtained by measuring the full-trace reuse distance. We first show the MRC in individual size classes of Facebook ETC workload. There are 32 size classes. The MRCs differ in most cases. Figure 9 shows three MRCs to demonstrate. The three curves have different shapes and positions in the plots, which means that data locality differs in different size classes. The shape of the middle curve is not entirely convex, which means that the traditional greedy solution, i.e. Stone et al. [26] in Section 6, cannot always optimize, and the dynamic-programming method in this work is necessary.

Figure 9 shows that the prediction is identical to the actual miss ratio for these size classes. The same accuracy is seen in all size classes. Table 2 shows the overall miss ratio of default Memcached for memory sizes from 128MB to 1024MB and compares between the prediction and the actual. The steady-state allocation prediction for default Memcached uses Equation 6 in Section 3.5. The prediction miss ratio uses Equation 4 based on predicted allocation. The actual miss ratio is measured from each run. The overall miss ratio drops as the memory size grows. The average accuracy in our test is 99.0%. The high MRC accuracy enables the effective optimization that we have observed in the last section.

5.4 LAMA Parameters

LAMA has two main parameters as explained in Section 3.4: the repartitioning interval M, which is the number of items accessed before repartitioning; and the reassignment upper bound N, which is the maximal number of reassignments at repartitioning. We have tested different values of M and N to study their effects. In this section, we show the performance of running the Facebook ETC workload with 512MB memory.

Figure 10 shows the dynamic miss ratio over the time. In all cases, the miss ratio converges to a steady state. Different M, N parameters affect the quality and speed of convergence. Three values of M are shown: 1, 2, and 5 million accesses. The smallest M shows the fastest convergence and the lowest steady-state miss ratio. They are the benefits of frequent monitoring and repartitioning. Four values of N are shown: 10, 20, 50, and 512. Convergence is faster with a larger N. However, when N is large, 512 in particular, the miss ratio has small spikes before it converges, caused by the increasing cost of slab reassignment. Frequent repartitioning speeds up the convergence but will increase the monitoring overhead.

In a real deployment, these two parameters can be adjusted according to the scale of the workload and Memcached server. Another design option is the size of reallocation unit. In our small scale test, every reassignment will move one slab from one class to another. However, in large scale deployments, a larger-grained reallocation unit leads to higher efficiency. It will bound the search space of Algorithm 1. Based on our experience on different-scale tests, the reallocation granularity can be set to r slabs, where r is the memory size of the Memcached server divided by 1GB. For example, if the size of cache server is 10GB, the unit of reallocation will increase correspondingly by 10 times to 10 slabs (10MB) for each reassignment. The reassignment upper bound N can be set to 50, so the total reassignments are limited to 50 times and each reassignment will move 10 slabs from one class to another. With this design, the number of solutions in the search space of Algorithm 1 is always bounded to $n * 1024 * 50$ (n is the number of classes). In our test machine, it only takes 0.01 seconds to determine the best allocation.

Now we discuss how to determine the repartitioning interval M in other cache sizes. With the reassignment upper bound, the interval should be long enough for each class to measure the miss ratio up to $N * r$ slabs above or below the current allocation. However, if the current allocation of some classes is smaller than the reassignment upper bound, miss ratio measurement up to this bound is meaningless. Meanwhile, doubling the current space allocation rarely happens in any class, so we choose to only measure miss ratios for up to 2 times of the current allocation. Assuming that the miss ratio of Class i is mr_i, and the probability of a reference to Class i is p_i, we can derive the following relationship:

$$M * p(i) >= \frac{min(2 * S_i, S_i + N * r) * I_i}{mr_i} \qquad (7)$$

where S_i is the current slabs allocation of Class i, I_i is the number of items in each slab of Class i. If we want to measure miss ratio curve of Class i, we use the number of distinct items that fill the desired slab allocation divides the current miss ratio to get the number of accesses to this class that should be monitored. With the above analysis, now we can determine the repartitioning interval M for a system of any scale:

$$M = \max_{1<i<n} \left(\frac{min(2 * S_i, S_i + N * r) * I_i}{mr_i * p(i)}\right) \qquad (8)$$

In real deployments, we can ignore those classes with few or no misses in Equation 8. Their current allocations are already sufficient to cache all the items being used. Miss ratio measurement for a larger allocation is not necessary.

5.5 Slab Calcification

LAMA does not suffer from slab calcification. Partly to compare with prior work, we use the three-phase workload (Section 5.1) to

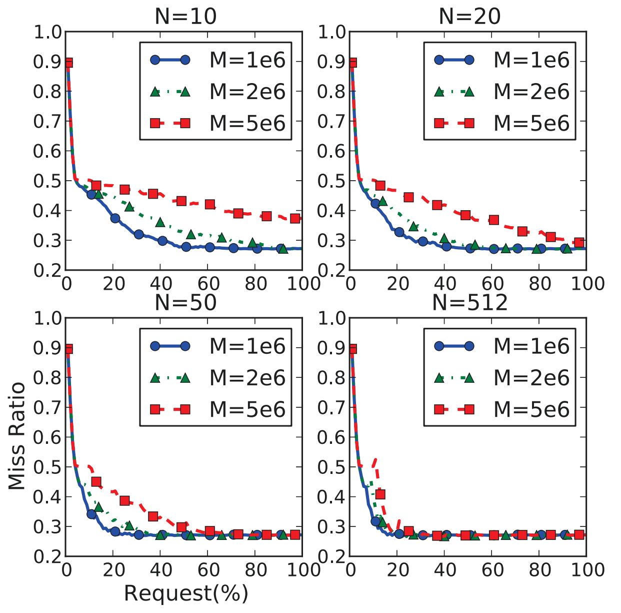


Fig. 10: Different combinations of the repartitioning interval M and the reassignment upperbound N

test how LAMA adapts when the access pattern changes from one steady state to another. The workload is the same as the one used by Carra et al. [14] using 1024MB memory cache to evaluate the performance of different strategies. Figure 11 shows the miss ratio over time obtained by LAMA and other policies. The two vertical lines are phase boundaries.

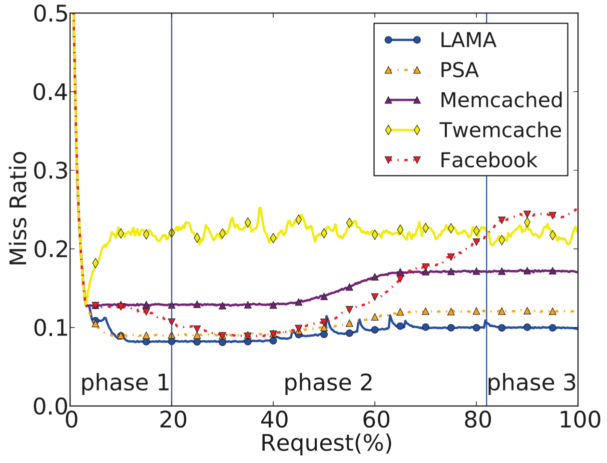


Fig. 11: Miss ratio over time by different policies

LAMA has the lowest miss ratio in all three phases. In the transition Phase 2, the miss ratio has 3 small, brief increases due to the outdated slab allocation based on the previous access pattern. The allocation is quickly updated by LAMA repartitioning among all size classes. In LAMA, the slabs are "liquid" and not calcified.

Compared with LAMA, the miss ratio of the default Memcached is about 4% higher in Phase 1, and the gap increases to about 7% in Phase 3, showing the effect in Phase 3 of the calcified allocation made in Phase 1. PSA performs very well but also sees its gap with LAMA increases in Phase 3, indicating that PSA does not completely eradicate calcification. Facebook uses global LRU. Its miss ratio drops slowly, reaches the level of PSA in Phase 2, and then increases fairly rapidly. The reason is the misleading LRU information when the working set changes. The items of the first set stay a long time in the LRU chain. The random eviction by Twemcache does not favor the new working set over the previous working set. There is no calcification, but the performance is significantly worse than others (except for the worst of Facebook).

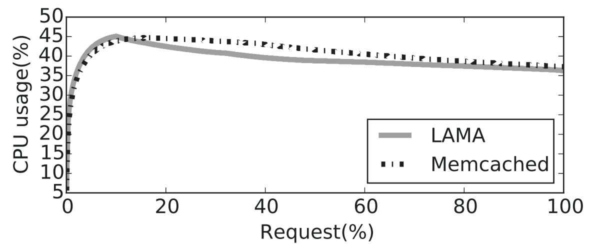


Fig. 12: Cpu usage over time by LAMA and default Memcached

To verify the CPU cost of LAMA. We monitor the CPU usage over time by LAMA and default Memcached in last test. For every 10^5 requests, the average CPU usage is recorded in Figure 12. As we can observe, LAMA has lower CPU cost than defalut Memcached for most of the time. In Figure 11, LAMA reduces the miss ratio compare to default Memcached, the CPU cost is also reduced because the querying and setting for missed data is less with lower miss ratio.

5.6 Theoretical Upper Bound

To measure the theoretical upper bound (TUB), we first measure the actual MRCs by measuring the full-trace reuse distance in the first run, compute the optimal slab allocation using Algorithm 1, and re-run a workload to measure the performance. The results for Facebook ETC were shown in Figures 7 and 8. The theoretical upper bound (TUB) gives the lowest miss ratio/ART and shows the maximal potential for improvement over the default Memcached. LAMA realizes 97.6% of the potential in terms of miss ratio and 92.1% in terms of ART.

We have also tested the upper bound for the three-phase workload. TUB shows the maximal potential for improvement over the default Memcached. In this test, LAMA realizes 99.2% of the potential in phase 3, while the next best technique, PSA, realizes 41.5%. At large memory sizes, PSA performs worse than the default Memcached. It shows the limitation of heuristic-based solutions. A heuristic may be more or less effective compared to another heuristic, depending on the context. Through optimization, LAMA matches or exceeds the performance of all heuristic solutions.

5.7 QoS Guarantee Server Partition

With the intra-server partition we proposed in Section 4, the desired memory space and slab allocation for performance-critical application can be calculated. Given a hit ratio requirement, LAMA can dynamically adjust the slabs partition to maintain the desired performance if the slab resources are sufficient. With the application's performance requirement increasing or declining, the number of slabs in the pool can be tuned for usage efficiency.

To demonstrate the effect of intra-server partition, we choose the three-phase workload ($APP1$) to run with Facebook ETC workload ($APP2$) in the same Memcached cluster. Because the hit ratio of $APP1$ drops from Phase 2 to Phase 3, it is reasonable to apply the dynamic intra-server partition to prevent the performance degradation. To balance the lengths of the two workloads, the ratio of their reference rates is set to 4:1. We evaluate the space occupation (Figure 13) and miss ratio (Figure 14) over time of two workloads under two configurations: (1) enabling LAMA without intra-server partition. All keys in both workloads are mapped to one Memcached server of 1024MB. (2) enabling LAMA and intra-server partition in the same time and make a hit ratio guarantee of $APP1$ at 85% and 80%. In the implementation, we set up two Memcached instances and limit their total memory to 1024MB. The initial memory space for each instance is 512MB.

In the first configuration ($APP1\_LAMA$), where the optimization target is the overall miss ratio of the server. LAMA dynamically adjusts the slab allocation for each size class. The two workloads freely compete for memory. The memory occupation varies as the access pattern changes. The first phase of $APP1$ reaches a hit ratio of 80%, but it drops to 78% after the phase transition. In the second configuration, the memory is partitioned at the beginning. To guarantee the hit ratio of $APP1$, every 10^7 accesses, we adjust the slabs partition using Algorithm 2. As the phase changes from 1 through 2 to 3, the slabs requirement of $APP1$ increases gradually. As we can observe, the performance degradation in the third phase is avoided. Except for a short miss ratio increase (up to 1.3%) in the transition stage of Phase 2, the miss ratio of the three phases maintains at the desired level. The performance of $APP2$ is sacrificed to satisfy the QoS requirement of $APP1$.

It should be pointed out that in the first half of Phase 2 the hit ratio of $APP1$ with shared but not-explicitly-partitioned memory and that with partitioned memory and 80%-QoS are very close. But the performance of $APP2$ during this time period is different. $APP2$ in partitioned memory shows a lower miss ratio while using less memory space. This demonstrates negative interference suffered by $APP2$ when sharing the memory. LAMA optimizes the slab allocation in shared memory according to the combined pattern of both workloads. For partitioned memory, $APP2$ is optimized only according to its own reference pattern. It performs better with an exclusive use of the memory. Memory partitioning brings benefits to $APP1$ (QoS guarantee) and $APP2$ (hit ratio) at the same time.

5.8 LAMA Scalability

In practice, memory demand of Memcached workloads can easily exceed the memory capacity of computer servers. For example, with 64GB memory on each server of Facebook's Memcached cluster, almost all memory is filled and a large number of misses are observed [17].

To evaluate the scalability of LAMA, we run the data caching benchmark program in the CloudSuite benchmark suite to simulate behavior of a Twitter caching server using a scaled Twitter dataset of 150GB. We set up two 32GB servers (64GB memory in total), each running a four-thread Memcached, to evaluate the performance under the heavy load of the benchmark. In this test, we are concerned only the differences of throughput and tail latency produced by large-scale Memcached with and without using LAMA. The data caching benchmark focuses on the stress

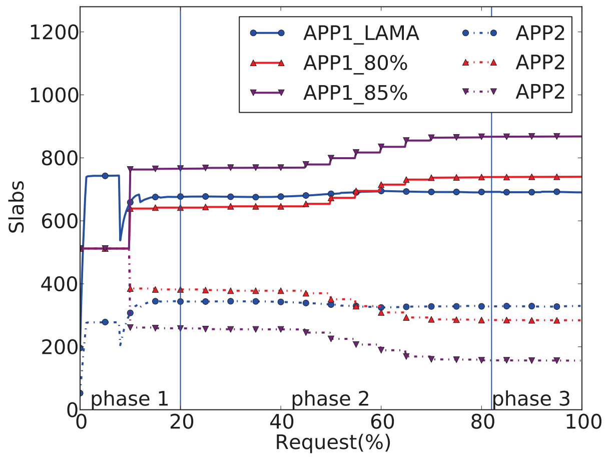


Fig. 13: Slabs occupation over time of different configuration

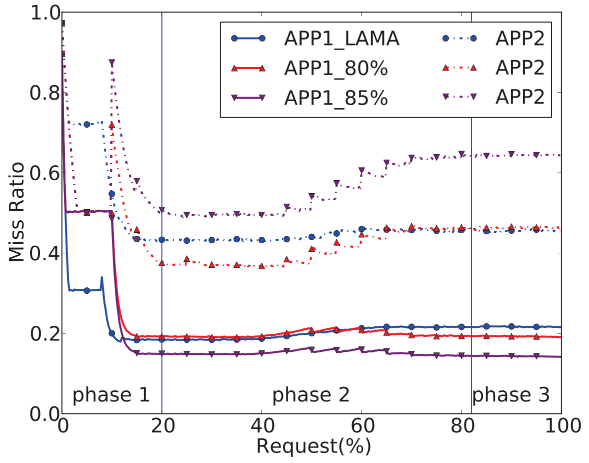


Fig. 14: Miss ratio over time of different configuration

test for Memcached. Upon every GET miss, it does not bring the missed data in cache. Since there is no data reuse, we cannot measure the miss ratio in this test and therefore cannot evaluate the benefit of LAMA. We use this experiment to evaluate the overhead of LAMA.

The benchmark is network-intensive. We use two 10Gbit Ethernet cards on each server. Two servers are configured with the same socket, but different IP address. We also run server-side Memcached and the client-side benchmark on different sockets of the same machine in order to remove the network bottleneck. After warming up the servers, we run the benchmark on the client side to test server performance with different number of threads. In the setup, there are 200 TPC/IP connections, and the GET/SET ratio is 0.8.

In Table 3, we adopt the most recent Memcached-1.4.29 with background garbage collector (automover) enabled to test the highest throughput and tail latency of the Memcached servers when LAMA is enabled or disabled. Under different request pressure, corresponding to number of threads used at the client, the Memcached servers with LAMA enabled exhibit performance similar to Memcached servers with LAMA disabled. Small differences between their throughput, average and tail latencies indicate small overhead of LAMA. In our test, higher workload pressure leads to higher request latency observed at the client. The peak

TABLE 3: The overhead of LAMA in large-scale Memcached severs reported by data caching benchmark of CloudSuite

	Threads	Throughput (Kreqs/sec)	Average Latency	90th	95th	99th	Min	Max
Disabled	8	373	7.3 ms	7.7 ms	7.8 ms	7.9 ms	6.4 ms	8.1 ms
Enabled	8	370	7.5 ms	7.9 ms	8.0 ms	8.1 ms	6.7 ms	8.2 ms
Disabled	16	383	16.4ms	26.8 ms	28.2 ms	30.2 ms	6.5 ms	33.8 ms
Enabled	16	378	16.6 ms	28.1 ms	30.7 ms	31.4 ms	6.8 ms	35.2 ms
Disabled	32	351	25.7 ms	28.2 ms	29.2 ms	31.0 ms	18.8 ms	33.2 ms
Enabled	32	343	27.1 ms	30.4 ms	32.1 ms	33.3 ms	20.2 ms	35.3 ms

throughput is observed when 16 threads, rather than 32 threads, are used at the client. The throughput degradation is due to the lower response time, which is caused by the inter-thread contention of Memcached server. As revealed in the experiment results, the MRC monitoring operations impose very limited overhead on the Memcached server. The overhead is especially small when clients and servers are not on the same machine and the network latency is dominant. LAMA's operations are performed only when the system is warming up or when the request pattern changes. Once the system reaches its optimal allocation, the MRC monitoring routine can stay idle incurring almost zero overhead until substantial performance change is detected.

To investigate how LAMA responds to slab calcification in a larger scale setup, we create a slab calcification scenario using the same twitter dataset as the one used in the pressure test. First, we warm up the Memcached server with the same dataset but increase each value size by a factor of 1.2. The purpose of this size increase is to create a different distribution of KV items across the size classes during cache warming up period. After the cache server is filled, we issue the original dataset with 100% GET commands and measure the miss ratio and throughput of the Memcached severs. This produces a slab calcification scenario, as the data pattern that determines the slabs allocation is different from the following access pattern exhibited after the cache warms up. We use libMemcached [27] to implement a multi-thread client to produce the same access pattern of CloudSuite. In order to measure the miss ratio of the system, we issue a SET command right after each GET miss to add the missed data into Memcached sever. To simulate the cache miss penalty, we add a 850ms latency with each GET miss. This latency is the average miss penalty observed in the experiment shown in Section 5.2, where it is caused by the back-end database query for re-generating the missed KV items.

Figure 15 shows the performance of Memcached servers (Memcached-1.4.29) before LAMA is enabled as well as the performance after calcification is resolved by LAMA. The Memcached servers with calcification produce a hit ratio of 79.5%. When LAMA is enabled and the optimal repartition is performed in both servers, the new partition delivers a hit ratio of 86.3%. Under different number of threads used by the client, the throughput of optimal partitioned Memcached servers is higher than the severs with calcification. Because higher hit ratio leads to smaller miss penalty, LAMA substantially improves system performance when calcification is observed. In this experiment, the reallocation granularity is 32 slabs. With the reassignment upper bound, the repartitioning can be completed within two seconds. It takes several such repartitioning operations before the system reaches its best performance. When LAMA is enabled, the additional space consumption is around 400MB in each server, which is 1.2% of entire memory space. When optimal partition is reached, LAMA can be disabled and the additional space can be freed.

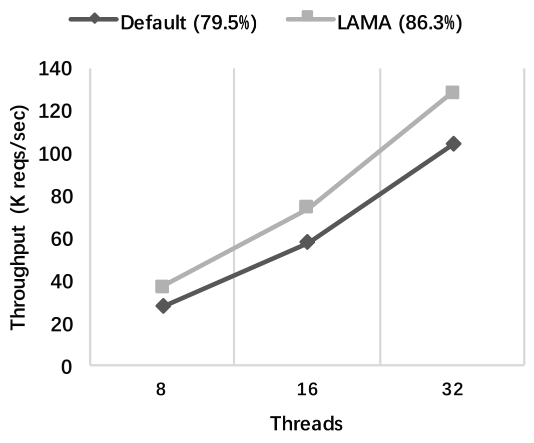


Fig. 15: The performance of Memcached servers with and without LAMA

6 Related Work

We have discussed related techniques on memory allocation in Section 2. Below we discuss additional related work in two other areas.

MRC Measurement Fine-grained MRC analysis is based on tracking the *reuse distance* or *LRU stack distance* [28]. Many techniques have been developed to reduce the cost of MRC profiling, including algorithmic improvement [29], hardware-supported sampling [30], [31], reuse-distance sampling [32], [33], [34], and parallel analysis [35], [36], [37]. Several techniques have used MRC analysis in online cache partitioning [38], [39], [31], page size selection [40], and memory management [41], [42]. The online techniques are not fine-grained. For example, RapidMRC has 16 cache sizes [31], and it requires special hardware for address sampling.

Given a set of cache sizes, Kim et al. divided the LRU stack to measure their miss ratios [42]. The cost is proportional to the number of cache sizes. Recently for Memcached, Bjornsson et al. developed MIMIR, which divides the LRU stack into variable sized buckets to efficiently measure the hit ratio curve (HRC) [6]. Both methods assume that items in cache have the same size, which is not the case in Memcached.

Recent work shows a faster solution using the footprint (Section 2.2), which we have extended in LAMA (Section 3.2). It can measure MRCs at per-slab granularity for all size classes with a negligible overhead (Section 5). For CPU cache MRC, the correctness of footprint-based prediction has been evaluated and validated initially for solo-use cache [16], [20]. Later validation includes optimal program symbiosis in shared cache [43] and a study on server cache performance prediction [44]. In Section 4.3, we have evaluated the prediction for Memcached size classes and shown a similar accuracy.

MRC-based Cache Partitioning The classic method in CPU cache partitioning is described by Stone et al. [26]. The method allocates cache blocks among N processes so that the miss-rate derivatives are as equal as possible. They provide a greedy solution, which allocates the next cache block to the process with the greatest miss-rate derivative. The greedy solution is of linear time complexity. However, the optimality depends on the condition that the miss-rate derivative is monotonic. In other words, the MRC must be convex. Suh et al. gave a solution which divides MRC between non-convex points [45]. Our results in Section 5.3 show that the Memcached MRC is not always convex.

LAMA is based on dynamic programming and does not depend on any assumption about MRC curve property. It can use any cost function not merely the miss ratio. We have shown the optimization of ART. Other possibilities include fairness and QoS. The LAMA optimization is a general solution for optimal memory allocation and partition. A similar approach has been used to partition CPU cache for performance and fairness [22], [19].

7 Conclusion

This paper has described LAMA, a locality-aware memory allocation for Memcached. The technique measures the MRC for all size classes periodically and reallocate the memory to reduce the miss ratio or the average response time. Compared with the default Memcached, LAMA reduces the miss ratio by 42% using the same amount of memory, or it achieves the same memory utilization (miss ratio) with 41% less memory. It outperforms four previous techniques in steady-state performance, the convergence speed, and the ability to adapt to phase changes. LAMA predicts MRCs with a 99% accuracy. As a result, its solution is close to optimal, realizing 98% of the performance potential in a steady-state workload and 99% of the potential in a phase-changing workload. LAMA has also been used to partition memory in a Memcached cluster to guarantee QoS for performance-critical applications.

Acknowledgments

The research is supported in part by the National Science Foundation of China (No. 61232008, 61272158, 61328201, 61472008 and 61170055); the 863 Program of China under Grant No.2012AA010905, 2015AA015305; the Research Fund for the Doctoral Program of Higher Education of China under Grant No.20110001110101; the National Science Foundation (No. CNS-1319617, CCF-1116104, CCF-0963759, CCF-0643664, CSR-1422342, CCF-0845711, CNS1217948).

References

[1] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[2] Shubin Zhang, Jizhong Han, Zhiyong Liu, Kai Wang, and Shengzhong Feng. Accelerating mapreduce with distributed memory cache. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 472–478. IEEE, 2009.

[3] Jinho Hwang, Ahsen Uppal, Timothy Wood, and Howie Huang. Mortar: filling the gaps in data center memory. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 53–64. ACM, 2014.

[4] Gurmeet Singh and Puneet Chandra Rashid Tahir. A dynamic caching mechanism for hadoop using memcached.

[5] Steven Hart, Eitan Frachtenberg, and Mateusz Berezecki. Predicting memcached throughput using simulation and modeling. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium*, page 40. Society for Computer Simulation International, 2012.

[6] Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[7] Kevin Lim, David Meisner, Ali G Saidi, Parthasarathy Ranganathan, and Thomas F Wenisch. Thin servers with smart pipes: designing soc accelerators for memcached. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, pages 36–47. ACM, 2013.

[8] Jithin Jose, Hari Subramoni, Krishna Kandalla, Md Wasi-ur Rahman, Hao Wang, Sundeep Narravula, and Dhabaleswar K Panda. Scalable memcached design for infiniband clusters using hybrid transports. In *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pages 236–243. IEEE, 2012.

[9] Bin Fan, David G Andersen, and Michael Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *NSDI*, pages 371–384, 2013.

[10] Jinho Hwang and Timothy Wood. Adaptive performance-aware distributed memory caching. In *ICAC*, pages 33–43, 2013.

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

[12] Caching with twemcache. https://blog.twitter.com/2012/caching-with-twemcache, 2014. [Online].

[13] Twemcache. https://twitter.com/twemcache, 2014. [Online].

[14] Damiano Carra and Pietro Michiardi. Memory partitioning in memcached: An experimental performance analysis. *Communications (ICC), 2014 IEEE International Conference on*, pages 1154–1159, 2014.

[15] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *NSDI*, pages 385–398, 2013.

[16] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 350–360. IEEE, 2011.

[17] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[18] Memcached-1.4.11. https://code.google.com/p/memcached/wiki/ReleaseNotes1411, 2014. [Online].

[19] Jacob Brock, Yechen Li, Chencheng Ye, and Chen Ding. Optimal cache partition-sharing : Dont ever take a fence down until you know why it was put up. robert frost. In *Proceedings of ICPP*, 2015.

[20] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. HOTL: a higher order theory of locality. In *ASPLOS*, pages 343–356, 2013.

[21] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[22] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. Recu: Rochester elastic cache utility – unequal cache sharing is good economics. In *Proceedings of NPC*, 2015.

[23] Shahram Ghandeharizadeh, Sandy Irani, Jenny Lam, and Jason Yap. Camp: a cost adaptive multi-queue eviction policy for key-value stores. In *Proceedings of the 15th International Middleware Conference*, pages 289–300. ACM, 2014.

[24] Mutilate. https://github.com/leverich/mutilate, 2014. [Online].

[25] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[26] Harold S Stone, John Turek, and Joel L. Wolf. Optimal partitioning of cache memory. *Computers, IEEE Transactions on*, 41(9):1054–1068, 1992.

[27] libmemcached. http://libmemcached.org/libMemcached.html, 2014. [Online].

[28] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.

[29] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.

[30] J Torrellas, Evelyn Duesterwald, Peter F Sweeney, and Robert W Wisniewski. Multiple page size modeling and optimization. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pages 339–349. IEEE, 2005.

[31] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. Rapidmrc: approximating l2 miss rate curves on commodity systems for online optimizations. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 121–132. ACM, 2009.

[32] Kristof Beyls and Erik H DHollander. Discovery of locality-improving refactorings by reuse path analysis. In *High Performance Computing and Communications*, pages 220–229. Springer, 2006.

[33] Derek L Schuff, Milind Kulkarni, and Vijay S Pai. Accelerating multicore reuse distance analysis with sampling and parallelization. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 53–64. ACM, 2010.

[34] Yutao Zhong and Wentao Chang. Sampling-based program locality approximation. In *Proceedings of the 7th international symposium on Memory management*, pages 91–100. ACM, 2008.

[35] Huimin Cui, Qing Yi, Jingling Xue, Lei Wang, Yang Yang, and Xiaobing Feng. A highly parallel reuse distance analysis algorithm on gpus. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1080–1092. IEEE, 2012.

[36] Saurabh Gupta, Ping Xiang, Yi Yang, and Huiyang Zhou. Locality principle revisited: A probability-based quantitative approach. *Journal of Parallel and Distributed Computing*, 73(7):1011–1027, 2013.

[37] Qingpeng Niu, James Dinan, Qingda Lu, and P Sadayappan. Parda: A fast parallel reuse distance analysis algorithm. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 1284–1294. IEEE, 2012.

[38] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing*, pages 1–12. ACM, 2001.

[39] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 89–102. ACM, 2009.

[40] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. Multiple page size modeling and optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, pages 339–349, 2005.

[41] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 177–188. ACM, 2004.

[42] Yul H Kim, Mark D Hill, and David A Wood. *Implementing stack simulation for highly-associative memories*, volume 19. ACM, 1991.

[43] Xiaolin Wang, Yechen Li, Yingwei Luo, Xiameng Hu, Jacob Brock, Chen Ding, and Zhenlin Wang. Optimal footprint symbiosis in shared cache. In *CCGRID*, 2015.

[44] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[45] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, 2004.

Xiameng Hu received his B.S. degree from Tianjin University in 2013.He is currently working towards the PhD degree in the school of electronics engineering and computer science at Peking University. His research interests include distributed computing, memory system optimization and data locality theory etc.

Xiaolin Wang received his B.S. and Ph.D. degrees from Peking University in 1996 and 2001 respectively. He is an associate professor in Peking University. His research interests include system software, virtualization technologies and distributed computing, etc.

Lan Zhou received his B.S. degree from Peking University in 2015. He is studying for a master's degree in Peking University. His research interests include system software, virtualization technologies and distributed computing, etc.

Yingwei Luo received his B.S. degree from Zhejiang University in 1993, and his M.S. and Ph.D. degrees from Peking University in 1996 and 1999 respectively. He is a professor in Peking University. His research interests include system software, virtualization technologies and distributed computing, etc.

Chen Ding received Ph.D. from Rice University, M.S. from Michigan Tech, and B.S. from Beijing University before joining University of Rochester in 2000. His research received young investigator awards from NSF and DOE. He co-founded the ACM SIGPLAN Workshop on Memory System Performance and Correctness (MSPC) and was a visiting researcher at Microsoft Research and a visiting associate professor at MIT. He is an external faculty fellow at IBM Center for Advanced Studies.

Song Jiang received his BS and MS degrees in computer science from the University of Science and Technology of China in 1993 and 1996, respectively, and received his PhD in computer science from the College of William and Mary in 2004. He is currently an associate professor at the Department of Electrical and Computer Engineering of Wayne State University. His research interests are in the areas of operating systems, file and storage systems, and high performance computing.

Zhenlin Wang received his BS degree in 1992 and MS degree in 1995 both in Computer Science and from Peking University, China. He received his PhD in Computer Science in 2004 from the University of Massachusetts, Amherst. He is currently a professor of the Department of Computer Science at Michigan Technological University. His research interests are broadly in the areas of compilers, operating systems and computer architecture with a focus on memory system optimization and system virtualization.