LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache

Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo
Peking University

Chen Ding
University of Rochester

Song Jiang
Wayne State University

Zhenlin Wang
Michigan Technological University

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 memory allocation. The default design does not make the best use of the memory. It fails to adapt when the demand changes, a problem known as *slab calcification*.

This paper introduces locality-aware memory allocation (LAMA), which solves the problem by first analyzing the locality of the Memcached requests and then repartitioning the memory to minimize the miss ratio and 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.

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 sustain a high hit rate.

Memcached [1] is a commonly used distributed in-memory key-value store system, which has been deployed in Facebook, Twitter, Wikipedia, Flickr, and many other internet companies. Some research also proposes 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 accepted 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.

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 modelling 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 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.

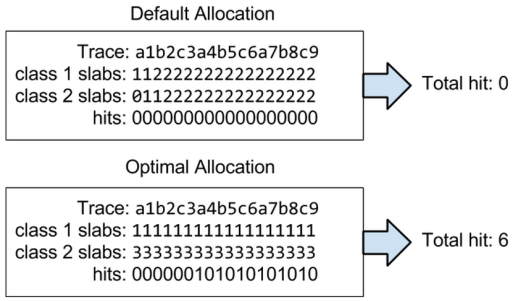
We propose a novel, dynamic slab allocation scheme, *locality-aware memory allocation* (LAMA), based on a recent advance on measurement of data locality [16] described in Section 2.2. This study provides a low-overhead yet accurate method to model data locality and generate miss ratio curves (MRCs). Miss ratio curve (MRC) reveals relationship between cache sizes and cache miss ratios. With MRCs for all classes, the overall Memcached performance can be modelled 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 (LAMA). The experimental results show LAMA can achieve over 98% of the theoretical potential.

2 Background

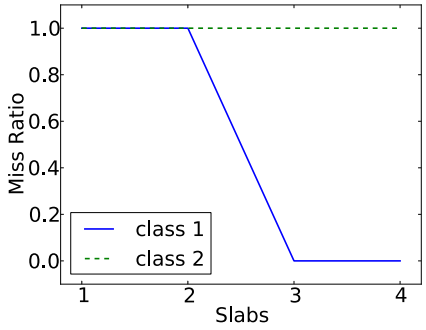
This section summarizes the Memcached's allocation design and its recent optimizations, which we will compare against LAMA, and a locality theory, which we will use in LAMA.

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 "*abcabcabc...*", and the sequence into Class 2 is "*123456789...*". 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 1 is 3, the hit ratio of Class 1 will be 100% with an allocation of 3 slabs according to the MRC in Figure 1(b). If we reallocate one slab from Class 2 to Class 1, the working set of Class 1 can be fully cached and every reference to Class 1 will be a hit. Although the hit ratio of Class 2 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.



(a) Access detail for different allocation



(b) MRCs for Class 1&2

Figure 1: Drawbacks of default allocation

Automove The open-source community has implemented an automatic memory reassignment algorithm (Automove) in a recent version of 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.

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 4. 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 quikly 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.

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].

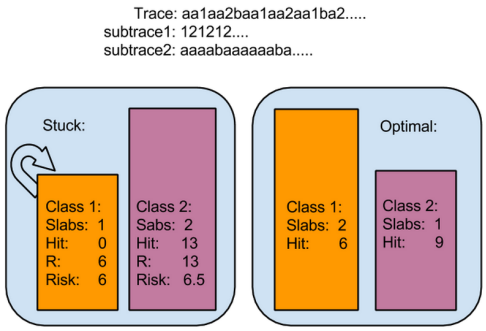


Figure 2: Drawbacks of PSA

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. The slabs are partitioned among size classes.

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 routine 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]:

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

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 smaller than x:

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

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:

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

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:

$$ART_i = MR_i * T_m(i) + (1 - MR_i) * T_h(i) \quad (4)$$

The overall ART of the Memcached server is:

$$ART = \frac{\sum_{i=1}^{n} R_i(ART_i)}{\sum_{i=1}^{n} R_i} \quad (5)$$

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 Repartitioning

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 repartitions 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 repartitioning 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 $M[i][j]$ when allocated j slabs, $0 \le j \le MAX$, 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:

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

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

$$Cost[i][j] \leftarrow (M[i][j] * T_m[i] + (1 - M[i][j]) * T_h[i]) * length(T[i])$$

Algorithm 1 Locality-aware Memory Allocation

Input: $Cost[][]$ // cost function, could be OPT_MISS or OPT_ART
Input: $S_{old}[]$ // number of slabs in each class
Input: MAX // total number of slabs

```
1:  function SLABREPARTITION(Cost[][], S_old[], MAX)
2:      F[][] ← +∞
3:      ▷ F[][] minimal cost for Class 1..i using j slabs
4:      for i ← 1..n do
5:          for j ← 1..MAX do
6:              for k ← 0..j do
7:                  Temp ← F[i−1][j−k] + Cost[i][k]
8:                                  ▷ Give k slabs to Class i.
9:                  if Temp < F[i][j] then
10:                     F[i][j] ← Temp
11:                     B[i][j] ← k
12:                       ▷ B[][] saves the slab allocation.
13:                 end if
14:             end for
15:         end for
16:     end for
17:     Temp ← MAX
18:     for i ← n..1 do
19:         S_new[i] ← B[i][Temp]
20:         Temp ← Temp − B[i][Temp]
21:     end for
22:     MR_old ← 0
23:     MR_new ← 0
24:     for i ← n..1 do
25:         MR_old ← MR_old + Cost[i][S_old[i]]
26:         MR_new ← MR_new + Cost[i][S_new[i]]
27:     end for
28:     if MR_old − MR_new > threshold then
29:         SlabReassign(S_old[], S_new[])
30:     end if
31: end function
```

Step 2: Repartitioning We design a dynamic programming algorithm to find new memory partitioning (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 pro-

cedure, LAMA reorganizes multiple slabs across all size classes. The dynamic programming algorithm is similar to Brock et al. [19] but for a different purpose.

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.

In order to avoid the cost of reassigning too many slabs, we set N slabs as the upper bound on the total reassignment. At each repartitioning, 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 re-allocation is global, since multiple candidate slabs are selected from possibly many size classes. In contrast, PSA selects a single candidate from one size class.

The bound N is the maximal number of slab reassignments. In the steady state, the repartitioning 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]. 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

We can also predict the performance of the default Memcached. Using Equation1 in Section 3.2, we can obtain the average footprint of any window size. 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 Evaluation

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

4.1 Experimental setup

LAMA Implementation We have implemented LAMA in Memcached-1.4.20. The implementation includes MRC analysis and slab reassignment. The MRC analysis is performed by a separate thread. Each analysis samples recent 20 million requests which are stored using a circular buffer. The buffer is shared by all Memcached threads and protected by a mutex lock for atomic access. During the analysis, it uses a hash table to record the last access time. The cost depends on the size of the items being analyzed. It is 3% - 4% of all memory depending for the workload we use. Slab reassignment is performed by dynamic programming as shown in Section 3.4. Its overhead is negligible, both in time and in space.

System Setup To evaluate LAMA and other strategies, we use a single node, Intel(R) Core(TM) I7-3770 with 4 cores, 3.4GHz, 8MB shared LLC with 16GB memory. The operating system is Fedora 18 with Linux-3.8.2. We set 4 server threads to test the system with memory capacity from 128MB to 1024MB. The small amount of memory is a result of the available workloads we could find (in previous papers as described next). In real use, the memory demand can easily exceed the memory capacity of modern systems. For example, one of our workloads imitates the Facebook setup that uses hundreds of nodes with over 64GB memory per node [17].

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. Memcached is running on local ports 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 [23], 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 *3-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 $\theta = 0$, $\phi = 312.6175$ and $k = 0.05$. The middle, transition phase increasingly accesses data objects from the second set.

- **A *stress-test* workload to measure the overhead.** We use the Memaslap generator of libmemcached [24], which is designed to test the throughput of a given number of server threads. Our setup follows Saemundsson et al. [6]: 20 million records with 16 byte keys and 32 byte values, and random requests generated by 10 threads. The proportion of GET requests to SET is 9:1, and 100 GETs are stacked in a single MULTI-GET request.

4.2 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 3 and 4 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.

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 4.5.

Next we compare the steady-state performance for memory sizes from 128MB to 1024MB in 64MB increments. Figures 5 and 6 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 4.6.

4.3 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 7 shows three MRCs to demonstrate. The

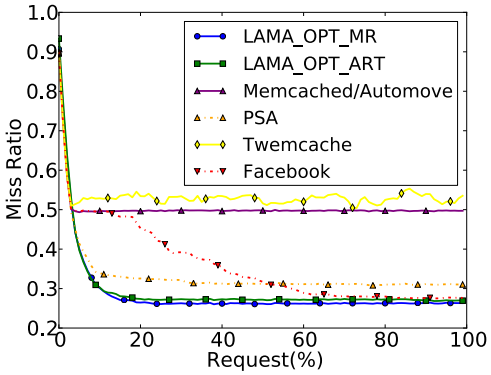


Figure 3: Facebook ETC miss ratio from cold-start to steady state

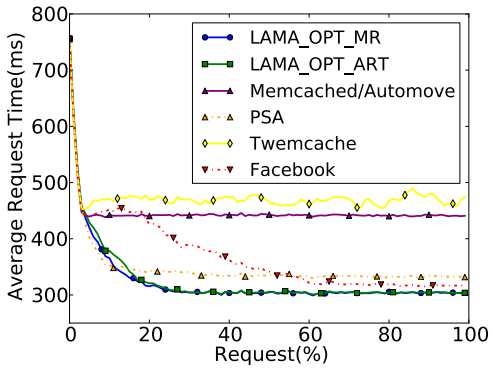


Figure 4: Average response time from cold-start to steady state

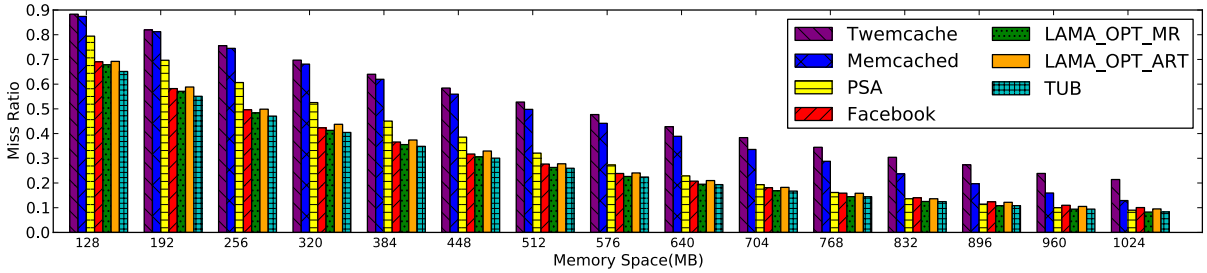


Figure 5: Steady-state miss ratio with different memory sizes

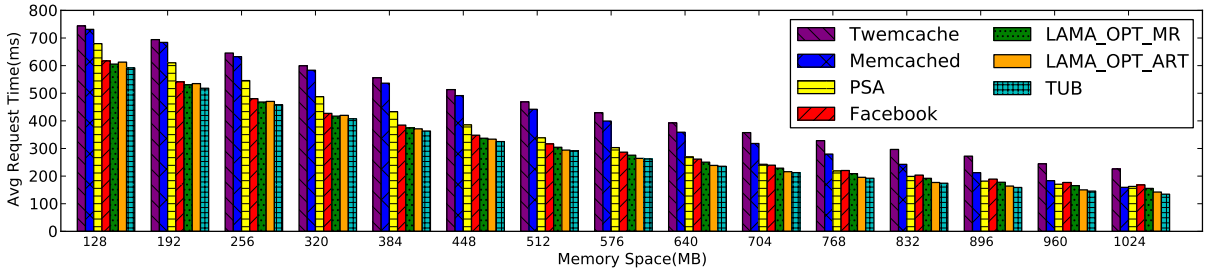


Figure 6: Steady-state average response time when using different amounts of memory

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. [25] in Section 5, cannot always optimize, and the dynamic-programming method in this work is necessary.

Figure 7 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 1 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.

4.4 LAMA Parameters

LAMA has two main parameters as explained in Section 3.4: the repartitioning interval M, which is the number of items accesses before repartitioning; and the reassignment upper bound N, which is the maximal number of slabs reassigned 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 Face-

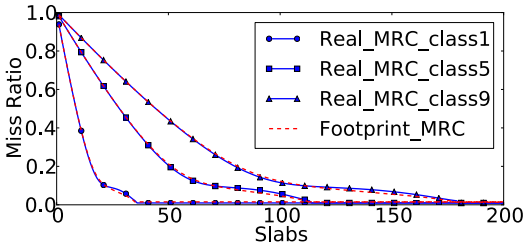


Figure 7: MRCs for class 1&5&9

Table 1: 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%

book ETC workload with 512MB memory.

Figure 8 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 especially, the miss ratio has small spikes before it converges, caused by the increasing cost of slab reassignment. For fast and steady convergence, we choose $M = 1,000,000$ *and* $N = 50$ for LAMA.

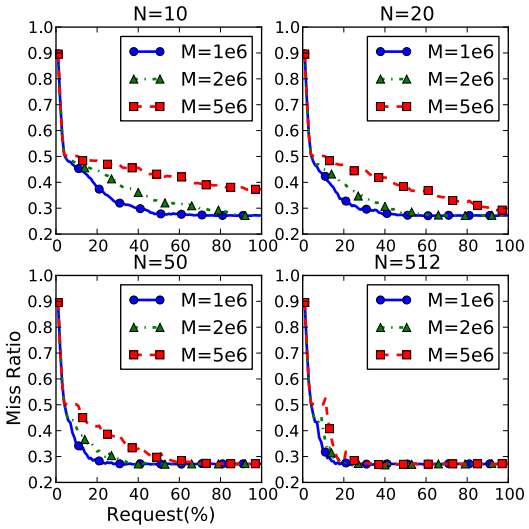


Figure 8: Different combinations of the repartitioning interval M and the reassignment upperbound N

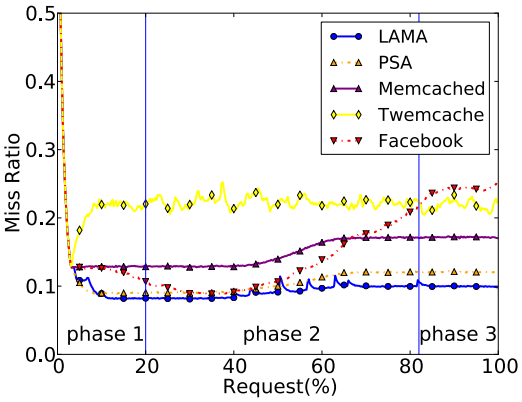


Figure 9: Miss ratio over time by different policies

4.5 Slab Calcification

LAMA does not suffer from slab calcification. Partly to compare with prior work, we use the 3-phase workload (Section 4.1) to 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 9 shows the miss ratio over time obtained by LAMA and other policies. The two vertical lines are phase boundaries.

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 "liquate" 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 perfor-

Table 2: Cost of MRC measurement in LAMA compared to reuse distance (RD)

Size class	Length (millions)	RD MRC (secs)	LAMA MRC (secs)	cost reduction
1	1.5953	3.6905	0.1159	96.85%
2	1.8660	4.5571	0.1378	96.97%
3	2.1091	5.2550	0.1597	96.96%
4	2.1140	5.3431	0.1598	97.00%
5	2.0646	5.2025	0.1554	97.01%
6	2.0875	5.2588	0.1585	96.98%
7	1.8725	4.6751	0.1404	96.99%
8	1.5546	3.7395	0.1131	96.97%
9	1.3022	3.0752	0.0932	96.96%

mance is significantly worse than others (except for the worst of Facebook).

4.6 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 5 and 6. 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 3-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.

4.7 LAMA Overhead

To be optimal, LAMA depends on accurate MRCs for all size classes at the slab granularity. In our implementation, we buffer and analyze 20 million requests before each repartitioning. In Table 2, we list the overhead of MRC measurement for Facebook ETC for the first 9 size classes. MRC based on reuse distance measurement (RD MRC), takes 3 to 5.4 seconds for each size class. LAMA uses the footprint to measure MRC. The cost is between 0.09 and 0.16 second, a reduction of 97% (or equivalently, 30 times speedup). In our experiments, the repartitioning interval is about 300 seconds. The cost of LAMA MRC, 0.1 second per size class, is acceptable for online use.

We have shown that LAMA reduces the average response time. A question is whether the LAMA overhead affects some requests disproportionally. To evaluate, we measure the cumulative distribution function (CDF) for the response time of LAMA and the default Memcached. The results are shown in Figure 10. The workload is ETC workload, and the memory size is 1024MB.

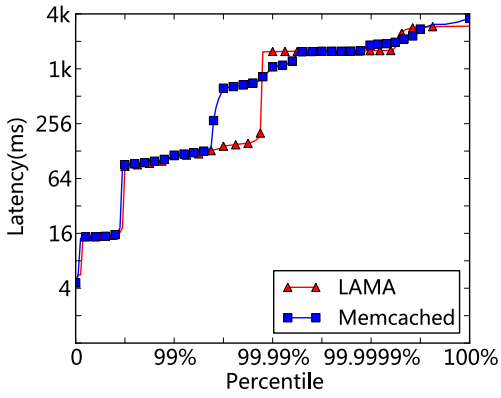


Figure 10: CDFs of request response latency

99.9% of the response times in LAMA are the same or lower than default Memcached. LAMA reduces the latency from over 512ms to less than 128ms for the next 0.09% requests. The latency is similar for the top 0.001% longest response times. The most significant LAMA overhead is the contention on the mutex lock when multiple tasks record their item access in the circular buffer. This contention and the other LAMA overheads do not cause a latency increase in the statistical distribution. LAMA's improved performance, however, reduces the latency by over 75% for 90% of the longest running requests.

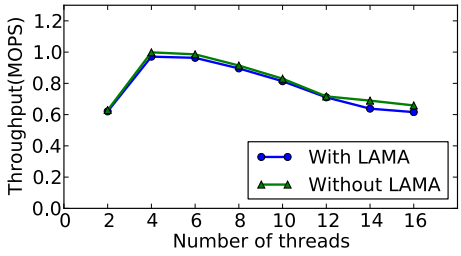


Figure 11: Throughput vs. number of threads

In this last experiment, we evaluate the throughput using stress test described in Section 4.1. The purpose is

to test the degradation when LAMA is activated. We repeat each test 10 times and report the average throughput. Figure 11 shows the overall throughput as different number of threads are used. Although the throughput of LAMA is lower than the default Memcached in the stress test, the average degradation is only 3.14%. In comparison, the Memcached performance profiler MIMIR [6], which we will introduce in Section 5, brings 8.8% degradation for its most accurate tracking. In actual use, LAMA is activated at the beginning and whenever the request pattern changes. Once LAMA produces the optimal partition, there is only the benefit and no overhead, as long as the system performance maintains stable.

5 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* [26]. Many techniques have been developed to reduce the cost of MRC profiling, including algorithmic improvement [27], hardware-supported sampling [28, 29], reuse-distance sampling [30, 31, 32], and parallel analysis [33, 34, 35]. Several techniques have used MRC analysis in online cache partitioning [36, 37, 29], page size selection [38], and memory management [39, 40]. The online techniques are not fine-grained. For example, RapidMRC has 16 cache sizes [29], 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 [40]. 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 4). 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 [41] and a study on server cache performance prediction [42]. 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. [25]. 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 [43]. Our results in Section 4.3 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 partitioning. A similar approach has been used to partition CPU cache for performance and fairness [22, 19].

6 Conclusion

This paper has described LAMA, a locality-aware memory allocation for Memcached. The technique measures the MRC for all size classes periodically and repartitions 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. 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.

7 Acknowledgements

We are grateful to Jacob Brock and the anonymous reviewers, for their valuable feedback and comments. 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, CNS-1217948).

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, Chencheng Ye, Chen Ding, Yechen Li, Xiaolin Wang, and Yingwei Luo. Optimal cache partition-sharing. 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] Mutilate. https://github.com/leverich/mutilate, 2014. [Online].

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

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

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

[42] 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.

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