mPart: Miss-Ratio Curve Guided Partitioning in Key-Value Stores

Daniel Byrne
Michigan Technological University
Houghton, Michigan, USA
djbyrne@mtu.edu

Nilufer Onder
Michigan Technological University
Houghton, Michigan, USA
nilufer@mtu.edu

Zhenlin Wang
Michigan Technological University
Houghton, Michigan, USA
zlwang@mtu.edu

Abstract

Web applications employ key-value stores to cache the data that is most commonly accessed. The cache improves an web application's performance by serving its requests from memory, avoiding fetching them from the backend database. Since the memory space is limited, maximizing the memory utilization is a key to delivering the best performance possible. This has lead to the use of multi-tenant systems, allowing applications to share cache space. In addition, application data access patterns change over time, so the system should be adaptive in its memory allocation.

In this work, we address both multi-tenancy (where a single cache is used for multiple applications) and dynamic workloads (changing access patterns) using a model that relates the cache size to the application miss ratio, known as a miss ratio curve. Intuitively, the larger the cache, the less likely the system will need to fetch the data from the database. Our efficient, online construction of the miss ratio curve allows us to determine a near optimal memory allocation given the available system memory, while adapting to changing data access patterns. We show that our model outperforms an existing state-of-the-art sharing model, *Memshare*, in terms of overall cache hit ratio and does so at a lower time cost. We show that for a typical system, overall hit ratio is consistently 1 percentage point greater and 99.9th percentile latency is reduced by as much as 2.9% under standard web application workloads containing millions of requests.

CCS Concepts • **Information systems → Cloud based storage**; *Hierarchical storage management*; *Web applications*;

Keywords Multi-tenant Architectures, Key-Value Stores, Miss Ratio Curves

ACM Reference Format:
Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: Miss-Ratio Curve Guided Partitioning in Key-Value Stores . In *Proceedings of 2018 ACM SIGPLAN International Symposium on Memory Management (ISMM'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3210563.3210571

ISMM'18, June 18, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5801-9/18/06…$15.00
https://doi.org/10.1145/3210563.3210571

1 Introduction

In-memory key-value stores play critical roles in many datacenters and web services by caching database requests. Two widely deployed systems are Memcached [23] and Redis [22]. Facebook lays claim to the largest reported deployment of Memcached [3], caching of the order of hundreds of terabytes of data. By the same token, Amazon Web services [2] hosts a large-scale deployment of Redis as a web application cache for its customers.

Given the scale and scope of these deployments it has been shown that even a slight improvement in hit ratio can have a significant impact on performance [3, 11]. Consider the following example: A cache has 98% hit ratio, average cache latency of 100us, and a database access time of 10ms, the expected end-to-end application latency is 298us (0.98 $\times$ 100us + 0.02 $\times$ 10000us). Now increasing the application's hit ratio to 99%, we get an expected end-to-end latency of 199us, resulting in over a 33% speedup.

In this work, we will focus on the multi-tenant environment, where there is a single caching instance (i.e. Memcached or Redis) partitioned for multiple applications (Figure 1). This environment has been promoted since it allows for pooling of memory between application to accommodate changing workloads [11]. In this environment we need a *sharing model* to decide how to allocate memory among the tenants as to not have one application take over the entire cache. Traditionally, the sharing model has been static, manual, assignments, but this often leads to underutilized memory and requires constant tuning [11].

A recent development in dynamic partitioning in multi-tenant environment has been to estimate the working-set size (WSS) of an application [13]. Ideally, the system should be adaptive to the working-set size of each application. With the recent advancements in miss ratio curve estimation algorithms, we study how to use full miss ratio curves to guide our memory partitioning between applications in order to deliver near optimal allocations [14, 15, 19, 20, 27].

Miss ratio curves (MRCs) relate cache allocations to application miss ratios. The MRC is an effective tool to estimate how an application responds to its cache allocation and *how*

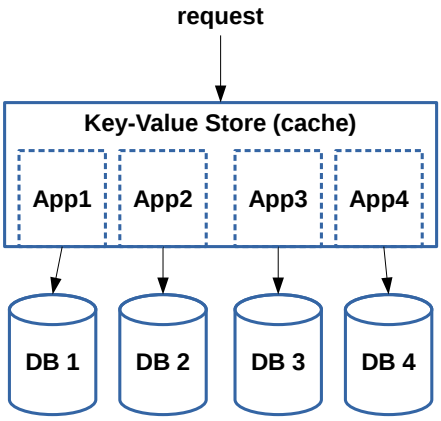

Figure 1. Example Multi-Tenant Architecture.

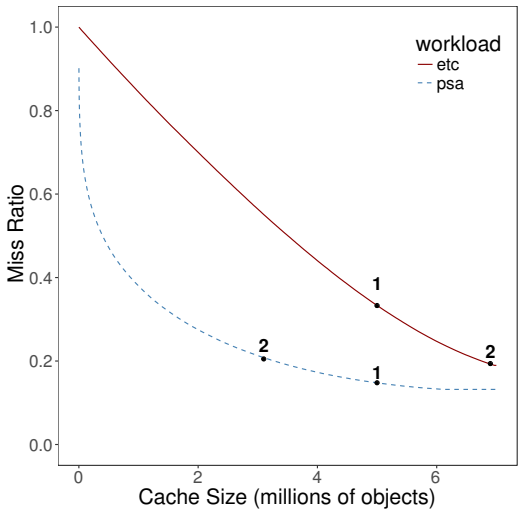

Figure 2. Example Miss Ratio Curve (MRC) illustrating the expected miss ratio for two allocations sharing a cache size of 10 million objects. Allocation 1 naively gives both applications 5 million objects, while Allocation 2 uses the miss ratio to maximize overall hit ratio.

much space it actually needs, known as the working set size. Knowing the working set size of an application helps estimate the *utility* of adding more memory or increasing the cache size. With MRC, previous studies usually define the WSS of an application as the minimum cache size that can achieve a preset hit ratio, say 95%, or the minimum cache size beyond which the hit ratio remains flat [30]. We choose the latter definition in this paper. Example miss ratio curves are shown in Figure 2.

Notice how that if our total cache size was 10 million objects, we can use the miss ratio curve to quickly find which allocation maximizes overall hit ratio. Compared to a naive assignment of 5 million objects to each application, a miss ratio curve based allocation gives 6.9 million objects to *etc* and 3.1 million objects to *psa*. This results in a 4 percentage point decrease in overall miss ratio (24% vs. 20%), or a 13% miss reduction.

In this paper, we design and implement *mPart*, a sharing model for multi-tenant key-value stores that allocates memory based on the miss ratio curve. We systematically compare mPart with *Memshare* (Section 2) [11], a state-of-the-art multi-tenant key-value store design that utilizes a credit system to incrementally adjust memory allocations. Our evaluation results show that mPart is able to provide a robust and complete picture of the working set versus Memshare and deliver higher hit ratio and smaller tail latency in response time at the 95th percentile and above.

2 Background

Figure 1 shows the multi-tenant environment. There is a single instance of the key-value store that is host to several applications. Each application has its own set of candidates for eviction, known as an *eviction pool*, and we assume there is no overlap in the keyspace.

Managing memory in a multi-tenant environment is not a new problem and there are several different *sharing models* available.

Static Allocation In larger deployments it is common to have several different *pools* of memcached servers. For example, Facebook partitions applications among a number of pools [3, 24]. Each pool is a set of memcached servers with varying levels of quality of service (QoS) needs. We must choose *which* applications belong in what pool in order to maximize overall performance. That is, if there is an application that would benefit the most from a pool with more memory then it should be placed in a pool with more memory. Currently this is done through static assignment and constant tuning.

Pooled Allocation Ideally, a cache should be able to adapt when an application's workload changes. In a pooled environment, all applications share the entire memory pool and eviction queues, essentially acting as a single application. As a greedy approach, it has the potential to starve smaller applications, but increases overall hit ratio compared to the static allocation strategy [11].

Working-Set Size Allocation Formally, researchers [1, 28] defined the multi-tenant allocation problem as the following optimization problem: compute the ideal memory $\mathbf{m} = [m_1, m_2, ..., m_N]$ that maximizes the number of hits in the cache,

Table 1. Example reuse time calculation for a set of requests

request	time	last use	reuse time
a	1	∞	∞
b	2	∞	∞
c	3	∞	∞
d	4	∞	∞
a	5	1	4
a	6	5	1
d	7	4	3
b	8	2	6

$$\begin{aligned} \underset{\mathbf{m}}{\text{maximize}} \quad & F(\mathbf{m}) = \sum_{n=1}^{N} (1 - mrc_i(m_i)) * NR_i \\ \text{subject to} \quad & \sum_{n=1}^{N} m_i \leq M \end{aligned} \tag{1}$$

for all N applications while subject to a memory budget M, where $mrc_i(m_i)$ is the miss ratio for a given memory size m_i, NR_i is the number of requests for the application, and the number of hits is $(1 - mrc_i(m_i)) * NR_i$.

Memory management has also been studied in the context of virtual machines running on a single host, where each virtual machine is allocated a certain amount of memory. Over time as workloads change, memory is reassigned based on the working set size of a given virtual machine [28].

2.1 Miss Ratio Curves

Miss ratio curves (MRCs) are useful for estimating *how much* data is being used by a particular workload and what utility can be gained by increasing the cache size. In our work, we will apply the recently developed MRC estimation method known as *AET: Average Eviction Time*.

Average Eviction Time (AET) Hu et al. present a series of kinetic equations related to average data eviction in the cache [19, 20]. The first step is to construct a histogram of *reuse time* that describe the reuse time distribution. The reuse time of an object is the *total* number of accesses between two accesses to the same piece of data. Reuse time is a simpler metric to measure than reuse distance [16, 32], which counts the number of distinct accesses between an access and its next reuse. Table 1 gives an example of reuse time calculation.

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

From here we can solve for the *average eviction time* for a given cache size, c. It is realized by the following equation:

$$\int_0^{AET(c)} P(t)dt = c \tag{2}$$

where $P(t)$ is the probability that a reference has reuse time greater than t. We obtain $P(t)$ from the reuse time histogram. Armed with the *average* eviction time for a cache of size c, we can now compute the MRC using the following equation described by Hu et al.:

$$mrc(c) = P(AET(c)) \tag{3}$$

where $P(AET(c))$ is the probability that a reuse time is greater than the average eviction time for cache of size c, $AET(c)$. The probability again comes from the reuse time histogram.

To control space and time overhead, AET adopts random sampling and reservoir sampling to generate reuse time histogram. We use random sampling in this paper. AET is able to generate an accurate MRC with a sampling ratio as low as 1/10000. In other words, it only needs to track the reuse time for 1 over 10000 accesses. AET maintains a small hash table of the sampled keys to compute the reuse time.

Cliffhanger [10] In contrast to computing the full miss ratio curve, another approach is to estimate the working set size by using *shadow queues*. Shadow queues are similar to victim caches [21], except that they only store the *keys* of the evicted items. The Cliffhanger miss ratio curve estimation uses shadow queues to hold the keys of evicted objects. If an application has a high hit ratio in the shadow queue, then it should be allocated more memory. If the cache *was* expanded by the size of the shadow queue, then those misses would have been hits. In addition, the more frequent an application is hitting in the shadow queue, the greater the gradient on the miss ratio curve is, since an increase in cache size would lead to significantly more hits. A credit system is used to keep track of which application *steals* memory from another application on a cache miss.

Memshare Cliffhanger is the sharing method that is implemented in the recent multi-tenant key-value store Memshare by Cidon et al [11]. On a miss in the cache, the Memshare system checks if the request would have been a hit in the application's shadow queue. If the application would have hit in the shadow queue, it is assigned 1 *credit* of memory. A credit is a pre-defined amount of memory that is taken from a victim application. The victim application is chosen based on *need*. An application's need is defined as the amount of memory *currently allocated* over the amount of *memory in use*. A need of less than 1 means the application is currently over-provisioned and therefore a good candidate to steal from. So, the Memshare system will prefer to take 1 credit from the application with the *lowest* need and assign it to application that hit in the shadow queue.

3 mPart: Miss Ratio Curve Guided Partitioning

A multi-tenant sharing model should be based on the working-set sizes of the applications present and must also be able to dynamically reassign memory in the face of changing workloads. To that end, we designed *mPart*. mPart is made up of two parts: online miss ratio curve construction and memory arbitration.

3.1 Miss Ratio Curve Construction

In order to determine the probability that a reference will be reused after a given time t, we must first construct a reuse time histogram. This is accomplished by sampling GET requests. When a request is randomly sampled, the algorithm checks a hash table of requests and access times in order to see if this particular request has been accessed before or not.

If the request has an entry in the hash table, then its reuse time is recorded in the reuse time histogram accordingly and its access time is updated as the current *logical time*. If there is no entry in the hash table for this request, then with a probability of the sampling ratio, the new request is added to the hash table and its *logical* access time is recorded.

Now that we have the reuse time histogram, we can calculate the miss ratio curve for each application by solving Equations 2 and 3. In the system implementation, this is done in a separate thread with the arbitration so it will not interrupt the key-value store service.

3.2 Arbitration

At a specified interval, we run the arbitration algorithm presented in [28]. Algorithm 1 shows the psuedocode. We use dynamic programming in order to minimize the number of expected misses based off the constructed miss ratio curve. In a set of N applications with total memory M, let $miss(i, j)$ be the minimum number of misses that the first i applications cause with total memory size j. So our goal is to find the assignment for $miss(N, M)$. $miss(i, j)$ can be calculated from the following recurrence equation using the application's miss ratio curve, mrc:

$$miss(i,j) = \min(miss(i,j), miss(i-1, j-k) + mrc_i(k) * NR_i) \quad (4)$$

where $L_i \leq k \leq H_i$. The L_i lower bound serves as a lower limit set by the administrator for the lowest amount of cache space required by the application. Similarly, the upper bound H_i is the maximum amount of cache space set by the administrator. In practice, we set up an increment/decrement step S for k. The step S determines the granularity of our miss ratio curve and therefore number of objects assigned to an application. In our evaluation, we found that 500 objects works well. We use NR_i for the number of accesses since the last arbitration. The base case is just the first application's misses, that is:

$$miss(1, j) = mrc_i(j) * NR_i \quad (5)$$

The expected memory allocation for application i is $E_i = \max(L_i, WSS_i)$. If $M \geq \sum E_i$, then we can distribute the additional, *bonus*, memory in proportion to an application's working set size. That is, if an application's working set size takes up 50% of $\sum E_i$, then it is given 50% of the additional memory.

If $M \leq \sum E_i$, then we cannot satisfy the working set size of at least one application, since our goal is to minimize the number of misses in the entire cache, we now need to calculate $miss(N, M)$ and record the assignments for each application. For each application (the i to N loop) we need to search over total memory sizes in M (the j loop). For each memory size j we need to find the memory allocation k (subject to the lower and upper bounds of the applications memory size), that minimizes the number of misses subject to j. The total time complexity is $O(N^2 \times (\frac{M}{S})^2)$.

4 Experimental Evaluation

In order to measure the memory allocation and performance of MRC guided partitioning and compare it with Memshare, we run two sets of tests. First, we run a set of 6 synthetic application traces in order measure the hit ratio gains and speedup. Second, we run the YCSB benchmark to measure the overhead of sampling and application arbitration.

Our experiments run on a 48-core 2.2GHz Intel(R) Xeon(R) CPU E5-2650 v5 and 256GB of DDR4 DRAM at 2400 MHz. All experiments are compiled and run on Red Hat 6.3.1 with Linux kernel 4.11.12 and GNU C compiler (gcc 7.3.0, -O3 optimization).

4.1 Memory Allocation Experiments

To measure the benefits of MRC guided partitioning, we adopted a set of synthetic workloads with the following: reuse patterns based on Zipfian-like distribution with varying α values to model web access patterns [6] and varying number of objects (ETC, PSA, YCSB) from [8, 12, 17]. Table 2 gives the parameters for each workload. Each workload has the same total number of requests: 100 million, and each object is 200 bytes. We define the working set size (WSS) as the number of objects in the cache such that any additional objects do not increase the cache hit ratio.

We use the log structured memory simulator from [11] to measure hit rates and compare against the Memshare system. We also vary the initial allocation, since it is possible developers do not have precise knowledge of the working set size of their application before deployment.

In order to compare allocation strategies, we use the workloads listed in Table 2 and vary the initial allocations. We set the following allocations:

Algorithm 1 Memory Allocation

Require: M ▷ Total cache memory
Require: {V} ▷ Set of applications
Require: {L} ▷ Set of lower limits for each app
Require: {H} ▷ Set of upper limits for each app
Require: {m} ▷ Set of current memory alloc's for each app
Require: {WSS} ▷ current WSS for each app
Require: {mrc} ▷ MRCs for each app
Require: {NR} ▷ requests for each app

procedure ARBITRATE
  for $i \in V$ **do**
    $E_i \leftarrow \max(low_i, WSS_i)$
  end for
  $E_s \leftarrow \sum E_i$
  if $M \geq E_s$ **then**
    $bonus \leftarrow M - E_i$
    for $i \in V$ **do**
      $A_i \leftarrow bonus \times \frac{E_i}{E_s}$
    end for
    return$\{A\}$
  else
    $Low_s \leftarrow 0, High_s \leftarrow 0$ ▷ bound on inner j loop
    for $i \in V$ **do**
      $Low_s \leftarrow Low_s + L_i$
      $High_s \leftarrow High_s + H_i$
      for $j \leftarrow Low_s; j \leq \min(M, High_s)$ **do**
        if $i = first$ **then**
          $miss[1][j] \leftarrow mrc_i(k) * NR_i$
        else
          $miss[i][j] \leftarrow \infty$
          for $k \leftarrow L_i; k \leq H_i$ **do**
            if $j \geq k$ **then**
              $cMiss \leftarrow mrc_i(k) \times NR_i$
              $pMiss \leftarrow miss[i-1][j-k]$
              $nMiss \leftarrow pMiss + cMiss$
              if $miss[i][j] \geq nMiss$ **then**
                $miss[i][j] \leftarrow nMiss$
                $Target[i][j] \leftarrow k$
              end if
            end if
            $k \leftarrow k + S$
          end for
        end if
      end for
    end for
    $T \leftarrow M$
    for $i \leftarrow N; i \rightarrow 1$ **do**
      $A_i \leftarrow Target[i][T]$
      $T \leftarrow T - Target[i][T]$
    end for
    return$\{A\}$
  end if
end procedure

Table 2. Workload parameters used. Number of objects is expressed in millions.

Workload	Unique objects	WSS	Zipf - α
etc	7	6.97	0.25
etc.small	3.5	3.47	0.25
psa	7	6.12	0.85
psa.small	3.5	3.38	0.85
ycsb	10	5.28	1.0
ycsb.small	5	3.63	1.0

Table 3. Initial allocations (millions of objects) used

Workload	90% WSS	75% Uniques	Equal
etc	6.273	5.25	4.3275
etc.small	3.123	2.625	4.3275
psa	5.508	5.25	4.3275
psa.small	3.042	2.625	4.3275
ycsb	4.752	7.5	4.3275
ycsb.small	3.267	3.75	4.3275
Total	25.965	27	25.965

- **90% Working Set** - Each application is initially given an allocation that is 90% of its working set size. This is to simulate when developers may have an estimation of the working set size of their application.
- **75% Unique Objects** - Each application is initially given 75% of its total unique objects. This is to simulate when developers may not be aware of the access pattern of the applications.
- **Equal Allocation** - All applications are given the same amount of memory. This is to simulate when developers do not have an estimation of each application's working set size. We set the total memory to be 90% of the sum of all applications working set size.

Table 3 compares each allocation and Figure 3 shows the miss ratio curve for each application.

Memshare Parameters The credit size is set to 500 objects based on the Memshare system defaults. The credit size is how much memory an application can steal from a low need application on a hit in the shadow queue. We found that an increase in credit size negatively impacted overall hit ratio for our set of experiments. We use a shadow queue size of 50K objects, based from previous work and our evaluation. We found that an increased shadow queue size of 20MB did not have significant impact on the hit ratio. Figure 4 shows the hit ratio and memory allocations of each application over the number of accesses.

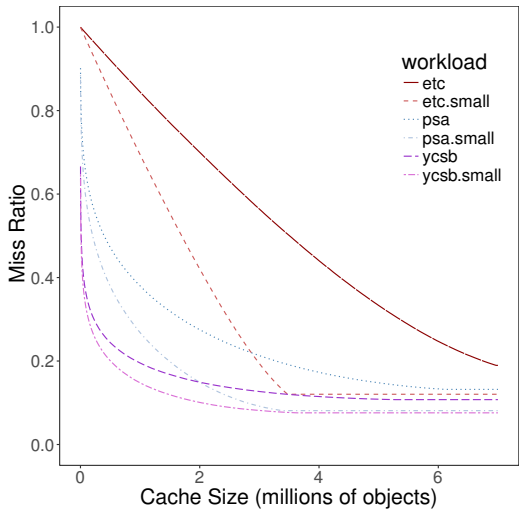

Figure 3. Miss Ratio Curve for each application used in our experiment

mPart Parameters We use a sample rate of 1/10,000 for building the reuse time histogram used by the AET algorithm. Our miss ratio curve is set to output at steps of 500 objects, $S = 500$, matching the credit size used in Memshare. The arbitration algorithm runs every 2 million accesses (50 times total for our experiments). We set an upper and lower bound of 30% for the amount of memory that can be added or removed from an application on a given arbitration.

Figure 4 shows the hit ratio and memory allocations of each application over the number of accesses. In all three settings, mPart outperforms Memshare since it has a complete miss ratio curve to calculate the utility of increasing or decreasing the memory of a given application. Table 4 summarizes the miss rates of each experiment. mPart reduces the miss ratio by at least 20% for all three settings and by over 25% in the 90% WSS allocation scenario.

90% Working Set Size Allocation In the working set size allocation, Memshare obtains a 95.2% hit ratio. The steady state is reached in the last 10% of accesses when allocations become relatively stable. Three applications, *psa*, *psa.small* and *ycsb.small*, ended up with similar allocations as their initial setting. The *etc.small* workload gained 400K objects in order to increase its hit ratio and the *etc* workload gained 100K objects. These objects came from the *ycsb* allocation, which resulted in an overall increase in hit ratio since *etc* and *etc.small* have steep miss ratio curves in comparison to *ycsb*.

mPart achieves a 96.4% hit ratio under its allocation performing repartitioning every 2 million accesses. The near-optimal tradeoff between cache size and miss ratio for applications like *etc* and *ycsb* is made in order to maximize hit ratio. For example, in the final *ycsb* allocation, we allocate 3 million objects (as opposed to the 4.1 million objects in Memshare) in order to achieve an increased hit ratio for *etc* and *etc.small*. While there is a decrease of 3.1% in *ycsb*'s hit ratio when compared with the Memshare allocation, the mPart allocation results in an overall 1.26% improvement in hit ratio or 25% miss reduction.

75% Unique Objects Allocation In the unique objects allocation, Memshare obtains a 95.8% hit ratio. This is the result largely due to the reassignment of the extra *ycsb* memory, decreasing from 7.5 million objects to 4.6 million objects. Initially, *ycsb* is overcommited, there are many more objects allocated than there are present in the cache. The need, $\frac{allocated}{present}$, is much less than 1, so it becomes an application that other applications like *etc* will steal from.

Since *ycsb* has such a large number of unique objects, it initially hits very little in the shadow queue. But at 58 million accesses, the shadow queue contains enough of the working set of requests to begin having enough hits to cause *ycsb* to steal from other applications. Unfortunately, this comes at the cost of other applications memory (*etc* or *etc.small*), which may have steeper miss ratio curves and not necessarily maximize overall hit ratio.

In comparison to the Memshare allocation, mPart achieves a 96.8% hit ratio, a 1 percentage point increase over the Memshare allocation. This is due to the increased allocation to applications *psa*, *psa.small*, and *etc*. And a decrease in allocation to applications *ycsb* and *ycsb.small*. Recall that a slight increase in hit ratio yields significant speedup. In our case from 95.8% to 96.8%, we should expect to see near 30% speedup assuming a cache access time of 10us and backend database access time of 10,000us.

Equal Objects Allocation In the equal objects allocation, Memshare obtains a 95.1% final hit ratio in the steady state. Again, we see a sub-optimal allocation choice in stealing memory from *etc* starting at 40 million accesses. This is because the application *psa* has higher need than *etc* as *etc* is currently over provisioned. Since *etc* has a steep miss ratio curve, the cost of stealing memory is significant. Before the reallocation *etc* had a hit ratio of nearly 100% but drops to 95% after the reallocation. Notice how if we were to steal the same amount of memory, about 200K objects, from the *ycsb* application, *ycsb*'s hit ratio would only decrease by a small amount based on the miss ratio curve (11.4% average miss ratio with cache size of 4.1 million to 11.6% average miss ratio with cache size 3.9 million).

In comparison to the Memshare allocation, mPart achieves a 96.1% hit ratio. Applications like *psa* and *etc* quickly receive close to their optimal memory allocation. This is one key advantage to using the stable miss ratio curve; we can almost immediately get an estimation of the working set size of an application. The resulting smaller allocations occur in order

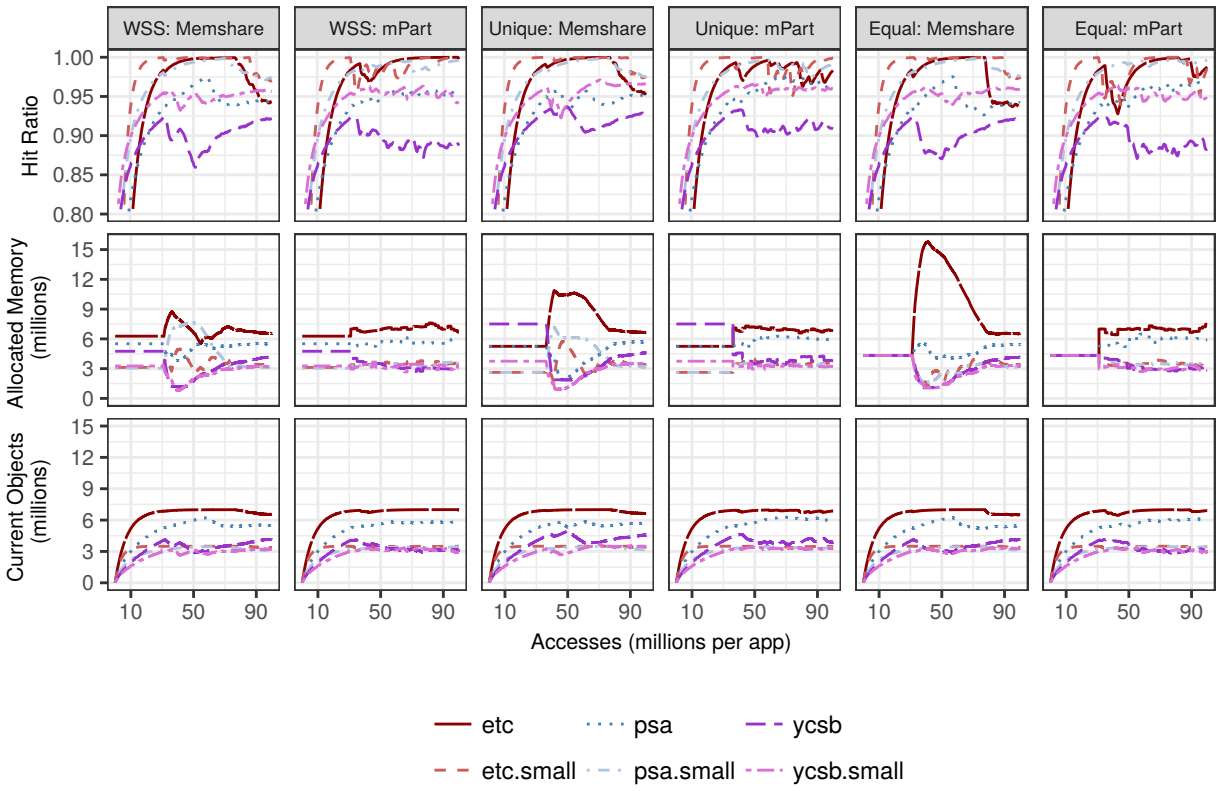

Figure 4. Hit ratio and Memory Allocation vs. Number of Accesses for each application

to accommodate the change in workloads. In comparison, the Memshare equal objects allocation test takes almost 80 million accesses to arrive at its working set size estimation.

4.2 Time-Varying Workload

In order simulate performance when workloads vary request rate over a period of time, as demonstrated in Facebook datacenter environment, we set the inter-arrival times for requests according to the observed patterns at Facebook. For each hour of the day, requests follow a Generalized Pareto distribution with varying shape parameter k, scale parameter σ, and constant threshold/location parameter $\theta = 0$ following Table 6 from B. Atikoglu et al. [3].

In our synthetic workloads we stagger the workloads by 4 hours each, so they do not follow the same periodic pattern. An example of this in the real world is when some applications see higher usage during the night time hours, such as bank batch processing, and others are used heavily during the daytime, such as web search.

Figure 5 shows that he overall performance is not changed significantly in mPart since the allocation assignments are unaffected by the different request rates because AET algorithm tracks *logical* access time. This is in contrast to Memshare since the hit rate in the shadow queue changes with the request rate, which in turn, drives the amount

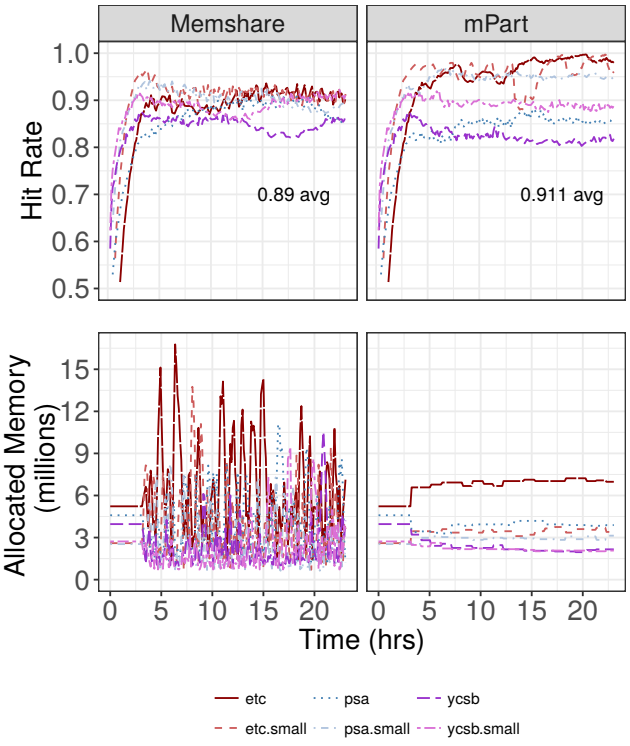

Figure 5. Time-Varying Workload over 24 hour period

Table 4. Summary of miss/hit ratio for each allocation

Miss / Hit Ratio (%)	90% WSS	75% Unique	Equal
Memshare	4.8 / 95.2	4.2 / 95.8	4.9 / 95.1
mPart	3.6 / 96.4	3.2 / 96.8	3.9 / 96.1
% change	**-25.0 / 1.26**	**-23.8 / 1.04**	**-20.4 / 1.05**

of memory that is stolen from another application. In Figure 5, we see that the allocations between applications in Memshare have considerably higher variance over time. The result is that mPart achieves a 2.4% improvement total hit rate over Memshare.

4.3 Noisy Neighbor Workload

Some applications have such poor locality that there may be little benefit from caching. Such applications are referred to as "noisy neighbors" and can eat away cache space from other applications if not controlled properly. To emulate a noisy neighbor application, we use a Zipfian-like distribution with $\alpha = 0.9$. We make 100,000 requests to a dataset of 100 million objects. The noisy neighbor enters the system at hour 5. The results from both Memshare and mPart are shown in Figure 6.

Both systems handle the noisy neighbor case well, since they both do estimations of an application's working-set size online. In the case of Memshare, the noisy workload rarely hits in the shadow queue, so it hardly ever steals memory from other applications. In the case of mPart, the miss ratio curve for the noisy neighbor is flat after a 150,000 objects. This means that there is no added benefit to increasing the noisy neighbor's cache space beyond that size. Figure 7 shows the miss ratio curve for the noisy neighbor workload. In fact, once the noisy workload enters the cache, both systems allocate it some memory (Memshare 4,000 and mPart 5,000 objects) but soon calculate that it is better to use the extra space towards workloads *psa* and *etc*. The noisy workload returns to its lower bound at hour 6 in the mPart system, while it takes an extra 2 hours for Memshare to realize this.

4.4 Miss Ratio Curve Accuracy

The AET model assumes a fully associative LRU cache. In the Memshare system cleaning is done based on application need. This means that after choosing a set number of *candidate* segments (in our experiments we choose 50 candidate segments), the system evicts items that belong to applications with *lower* need first. Within an application, items are evicted on LRU ordering. While not completely LRU, it has been shown that sampling for eviction is accurate and yields lower overhead [5].

In order to measure the actual miss ratio for a given cache size, we ran Memshare for a range of cache sizes, 200K to 8 million objects, and recorded the miss ratio for each run.

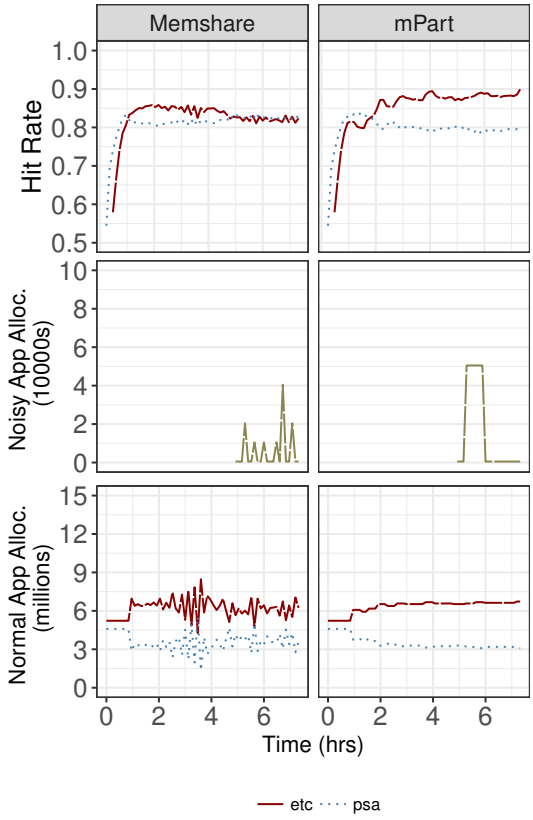

Figure 6. Applications with a noisy neighbor added

Each workload was run separately so there was no sharing to impact the miss ratio. The error in the AET model prediction is shown in Figure 8. It provides accurate results at a sampling rate at 1/10,000 requests.

4.5 Sampling and Arbitration Overhead

To measure the sampling and arbitration overhead, we implement our approach on the existing Memshare system. The clients and cache server run on one machine, hence the measurements represent the worst case. We use the YCSB framework using 250 byte items with 23 byte keys over 100 million operations. The access pattern is the YCSB zipf distribution. The results are the average of 10 runs.

We use a sampling rate of 1/10,000 and implement the arbitration into the log cleaner in order to avoid unnecessary slowdown since the cleaner runs in a separate thread. Our

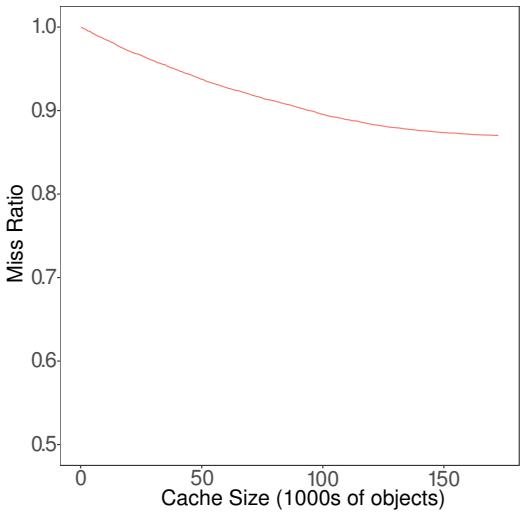

Figure 7. Noisy neighbor MRC

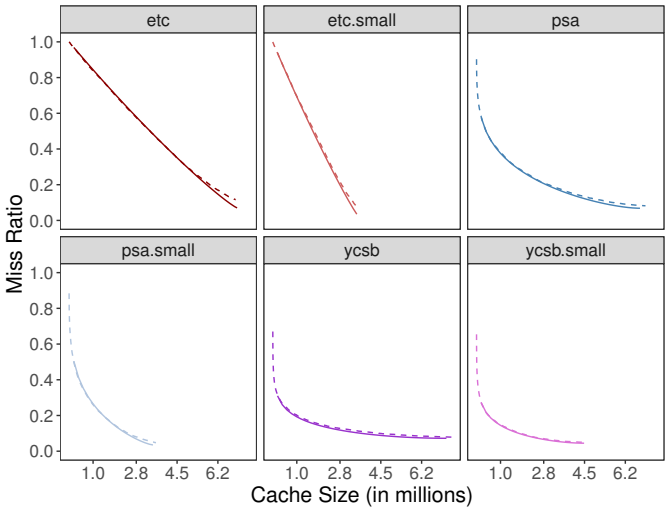

Figure 8. Accuracy of MRC generated by AET vs. actual

results show a significant decrease in tail latency since we do not have to check if a request would have been a hit in the shadow queue on every miss.

In mPart, we have a 1/10,000 chance of adding a request's reuse time to the reuse time histogram. In the Memshare scheme, checking the shadow queue happens on every miss. The extra overhead of checking the shadow queue much more often than mPart is reflected in the tail latencies show in Tables 5 and 6.

CPU Utilization and Throughput In terms of CPU utilization, mPart achieves lower cleaning costs compared to Memshare because we do not need to push cleaned items into their respective shadow queues. Table 7 compares the CPU time spent on different tasks. In mPart, we solve the AET equation off the critical path in the cleaner thread. Our results show that solving the AET equation is cheaper than maintaining shadow queues. For sampling, we find that only 0.002% of CPU time is spent for sampling, while up to 1.1% of CPU time is spent testing shadow queues on GET misses in Memshare.

Table 8 shows the increased throughput of mPart compared with Memshare. This is a result of 1) increased hit ratio and 2) lower latency.

Memshare and mPart both have relatively small space overheads; on the order of 100s of kilobytes. For Memshare, the shadow queue represents 10MB (or 20MB) of items. Our tests have an item size of 200 bytes, so one queue stores 50,000 keys. Since we only keep the 8 byte key hashes, that leaves a total overhead of 400KB (or 800KB if using 20MB queue) per application. In our experiments increasing the shadow queue size did not increase the hit ratio of the total system. For mPart, we consider the space overhead of AET algorithm. The reuse time histogram is by default 100,000 entries, which gives 800KB space overhead. The hash table that keeps samples of reuse times is determined by the number of references being tracked at a given time. The upper bound on the hash table is then, the working set size times the sampling rate (1/10,000 for this work). In our case, the *etc* workload has the largest WSS of 6.97 million objects, so it's hash table needs to hold at most 697 objects, at 8 bytes each entry, resulting in 5.4KB of space for this application.

5 Related Work

Other sub-linear time miss ratio curve construction techniques include SHARDS [27], Counter-Stacks [29], StatCache [4] and StatStack [15]. SHARDS starts by spatially sampling accesses by using hashed value of their location and then stores the samples in an interval tree for fast reuse distance computation, which then can be used to construct the MRC.

Counter-Stacks uses probabilistic structures like, HyperLogLogs and Bloom filters, in order to maintain a list of unique accesses over a period of time, while maintaining low overhead. The list of unique accesses allows for keeping track of how much a counter has been increased over a number of non-distinct accesses. This gives the stack distance, which then is used to calculate the miss ratio curve.

StatCache and StatStack also sample the *reuse times* of references in a similar way to AET. The Statcache model assumes a cache with random replacement, because of this we can assume that the probability that a cache line is still in a cache after a cache miss is uniform. Their model calculates the probability that a reference will cause a cache miss by fixing the miss ratio. Then the probability that a given number of references will cause a miss can be related to the references' reuse time. This differs from AET since AET calculates the average eviction time based on the reuse time distribution.

Table 5. GET latencies measured in μs

GETS	50%	75%	90%	95%	99%	99.5%	99.9%
Memshare	29.46	32.06	35.41	38.27	45.13	47.4	55.41
mPart	29.36	31.89	35.15	37.42	44.48	46.86	53.81
% change	**-0.34**	**-0.53**	**-0.73**	**-2.22**	**-1.44**	**-1.14**	**-2.89**

Table 6. SET latencies measured in μs

SETS	50%	75%	90%	95%	99%	99.5%	99.9%
Memshare	34.2	38.23	43.4	48.69	85.99	95.93	126.69
mPart	34.06	38.06	43.21	47.98	82.92	94.05	123.77
% change	**-0.41**	**-0.44**	**-0.44**	**-1.46**	**-3.57**	**-1.96**	**-2.30**

Table 7. Percentage of CPU time spent on cleaning and online WSS tracking. Cleaning involves maintaining shadow queue for Memshare and solving AET equation for mPart. WSS tracking involves checking shadow queue for Memshare and sampling references for mPart.

CPU Time	Cleaning	WSS Tracking
Memshare	4.4%	1.1%
mPart	3.2%	0.002%

Table 8. Throughput (1000 op/s) for Memshare and mPart.

Throughput	YCSB workload
Memshare	560.1
mPart	610.8
% change	**8.8**

The Stackstack model builds off of the Statcache work by modeling an LRU cache. They define the expected stack distance of a reference with reuse time t to be the average stack distance of all references with reuse time t. The miss ratio curve is then constructed by computing the expected stack distance of each reuse time weighted by their frequency, this gives us a stack distance distribution. Stackstack performs in the same time bound as the AET model, $O(N)$ time and $O(1)$ space due to their sampling techniques.

Miss ratio curves have been used in *memcached* for provisioning slab classes. In memcached each object belongs to a class based on its size, and so MRCs can be used to maximize hit ratio over all classes for a given amount of memory. LAMA [17, 18] achieves this by using the *footprint* metric introduced by Ding et al [14]. Mimir [26] and Dynacache [9] also approach the problem by modeling the miss ratio for each slab class.

When applications share data, FairRide [25] provides a model that provides near-optimal fairness in cache memory allocation. In our work, applications have their own keyspace, which is consistent in Facebook [3] and Memcachier [11] deployments.

In the context of virtualization, memory allocation between virtual machines on a single host has been studied extensively [31]. mPart uses the dynamic programming algorithm presented by Wang et al. [28] for its memory balancer. The memory balancer uses the miss ratio curve in order to minimize the expected number of page misses for a given application. This is in turn used to guide virtual machine memory allocation in a cloud environment.

6 Conclusion

We have demonstrated that in a multi-tenant caching environment, full miss ratio curve guided partitioning outperforms the existing state-of-the-art in terms of hit ratio and latency. mPart serves as an initial sharing model that can be applied to other multi-tenant environments where memory is limited.

Future work can improve on the arbitration (when to run) and other methods of solving the optimization problem (i.e. LP solvers). In addition, within multi-tenant environments there is often a heterogeneous backend. For example, an application may access a database that runs on solid-state flash storage or a database that resides on traditional hard disk.

7 Availability

Our modified version of *lsm-sim*, which includes the AET miss ratio curve implementation and arbitration can be found here:

`https://github.com/mpart-mtu/lsm-sim`

Acknowledgments

We would especially like to thank Ryan Stutsman and Asaf Cidon for providing us with the Memshare system, the log-structured memory simulator (*lsm-sim*), and their correspondence throughout this work. This research is supported in part by the National Science Foundation under Grant No. CSR1618384 and CSR1422342, the National Science Foundation of China under Grant No. 61232008, 61472008, 61672053 and U1611461, Shenzhen Key Research Project under Grant No. JCYJ20170412150946024, and the 863 Program of China under Grant No. 2015AA015305. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

References

[1] Cristina L. Abad, Andres G. Abad, and Luis E. Lucio. 2017. Dynamic Memory Partitioning for Cloud Caches with Heterogeneous Backends. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 87–90.

[2] Amazon. 2018. Amazon Web Services. (May 2018). Retrieved May 10, 2018 from https://aws.amazon.com

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64.

[4] E. Berg and E. Hagersten. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *2004 IEEE International Symposium on - ISPASS Performance Analysis of Systems and Software (ISPASS '04)*. 20–27.

[5] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 499–511.

[6] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. 1999. Web caching and Zipf-like distributions: evidence and implications. In *Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings (INFOCOM '99)*, Vol. 1. 126–134 vol.1.

[7] Damiano Carra. 2016. Benchmarks for testing memcached memory management. (December 2016). Retrieved December 10, 2016 from http://profs.sci.univr.it/~carra/mctools/

[8] D. Carra and P. Michiardi. 2014. Memory partitioning in Memcached: An experimental performance analysis. In *2014 IEEE International Conference on Communications (ICC '14)*. 1154–1159.

[9] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA.

[10] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 379–392.

[11] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 321–334.

[12] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 143–154.

[13] Peter J. Denning. 1968. The Working Set Model for Program Behavior. *Commun. ACM* 11, 5 (May 1968), 323–333.

[14] Chen Ding and Xiaoya Xiang. 2012. A Higher Order Theory of Locality. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC '12)*. ACM, New York, NY, USA, 68–69.

[15] D. Eklov and E. Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS '10)*. 55–65.

[16] Changpeng Fang, S. Can, S. Onder, and Zhenlin Wang. 2005. Instruction based memory distance analysis and its application to optimization. In *14th International Conference on Parallel Architectures and Compilation Techniques (PACT'05)*. 27–37.

[17] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 57–69.

[18] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. 2017. Optimizing Locality-Aware Memory Management of Key-Value Caches. *IEEE Transactions on Computers (TC '17)* 66, 5 (May 2017), 862–875.

[19] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 351–364.

[20] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Trans. Storage (TOS'18)* 14, 2, Article 12 (April 2018), 34 pages.

[21] N. P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *The 17th Annual International Symposium on Computer Architecture (ISCA '90)*. 364–373.

[22] Redis Labs. 2018. redis. (May 2018). Retrieved December 10, 2016 from https://redis.io

[23] memcached. 2018. memcached. (2018). Retrieved May 10, 2018 from https://memcached.org

[24] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 385–398.

[25] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. 2016. FairRide: Near-Optimal, Fair Cache Sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 393–406.

[26] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. 2014. Dynamic Performance Profiling of Cloud Caches. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '14)*. ACM, New York, NY, USA, Article 28, 14 pages.

[27] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Berkeley, CA, USA, 95–110.

[28] Zhigang Wang, Xiaolin Wang, Fang Hou, Yingwei Luo, and Zhenlin Wang. 2016. Dynamic Memory Balancing for Virtualization. *ACM Trans. Archit. Code Optim. (TACO '16)* 13, 1, Article 2 (March 2016), 25 pages.

[29] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. 2014. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield,

CO, 335–349.

[30] Ting Yang, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2006. CRAMM: Virtual Memory Support for Garbage-collected Applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 103–116.

[31] Weiming Zhao and Zhenlin Wang. 2009. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '09)*. ACM, New York, NY, USA, 21–30.

[32] Yutao Zhong, Xipeng Shen, and Chen Ding. 2009. Program Locality Analysis Using Reuse Distance. *ACM Trans. Program. Lang. Syst. (TOPLAS '09)* 31, 6, Article 20 (Aug. 2009), 39 pages.