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

Daniel Byrne
Michigan Technological University, djbyrne@mtu.edu

MPART: MISS RATIO CURVE GUIDED PARTITIONING IN KEY-VALUE STORES

By
Daniel Byrne

A THESIS
Submitted in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE
In Computer Science

MICHIGAN TECHNOLOGICAL UNIVERSITY
2018

This thesis has been approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE in Computer Science.

Department of Computer Science

Thesis Co-advisor:	*Dr. Zhenlin Wang*
Thesis Co-advisor:	*Dr. Nilufer Onder*
Committee Member:	*Dr. Soner Onder*
Department Chair:	*Dr. Zhenlin Wang*

Dedication

To my mother, father, teachers and friends who have always encouraged me to study and pursue my education.

Contents

List of Figures

List of Tables

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.

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 thesis, 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 cache hit ratio and does so at a lower time cost. We show that average 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.

Chapter 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 [33] and Redis [31]. Facebook lays claim to the largest reported deployment of memcached [5], caching of the order of hundreds of terabytes of data. By the same token, Amazon Web services [4] 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 [5, 17]. 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 thesis, we consider the multi-tenant environment, where there is a single caching instance (i.e. memcached or Redis) partitioned for multiple applications (Figure 1.1). This environment has been promoted since it allows for pooling of memory between application to accommodate changing workloads [17]. In this environment we need a *sharing model* to decide how to allocate memory among the tenants as to not have a few applications 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 [17].

A recent development in dynamic partitioning in multi-tenant environment has been to estimate the working-set size (WSS) of an application [20]. Ideally, the system

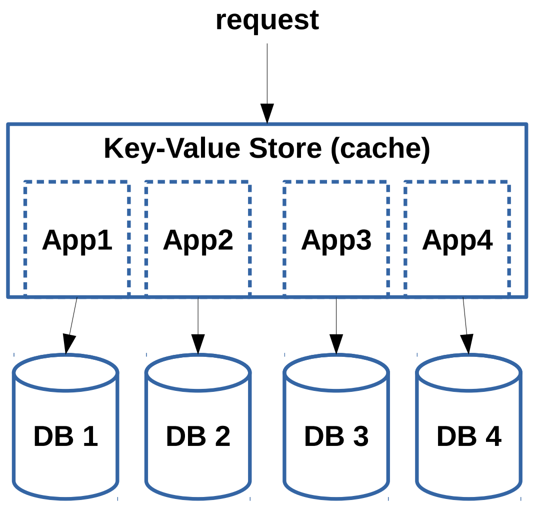


Figure 1.1: Example Multi-Tenant Architecture. There is a single cache that serves requests to multiple applications/databases, space is shared between each application.

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 [21, 22, 27, 28, 42].

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 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 [46]. We choose the latter definition in this paper. Example miss ratio curves are shown in Figure 1.2.

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 an 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.

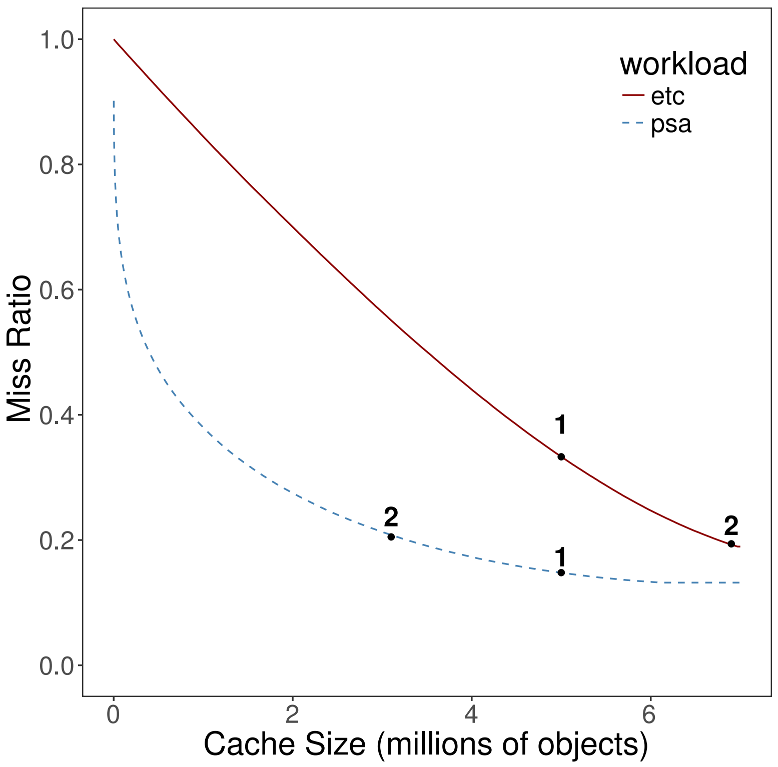


Figure 1.2: Example Miss Ratio Curve (MRC) illustrating the expected miss ratio for two allocations sharing a cache size of 10 million objects. Allocation 1 gives both applications 5 million objects, while Allocation 2 uses the miss ratio to maximize overall hit ratio.

In this thesis, we present the implementation of *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* (Chapter 2) [17], 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.

Chapter 2

Background

In this chapter we will describe key-value stores in detail and provide examples of their use in a typical environment. We also motivate the multi-tenant design in the data center environment based off of prior work in the virtual machine world and theoretical results on resource sharing. In doing so, we present different *sharing models* for how to divide the memory available among the applications hosted on a single instance of a key-value store.

Finally, we describe the recent work surrounding working set modeling and how they are used to make decisions about allocating memory among multiple applications. In particular, we describe an incremental approach, *Cliffhanger*, which uses a victim cache to estimate the gradient of the miss ratio curve. We also present an overview of a complete miss ratio curve algorithm, Average Eviction Time (AET), which uses the logical reuse time of references.

2.1 Key-Value Stores

Key-Value stores are a class of object stores that store data in a `<key,value>` pair. Typically keys are represented as integers or strings and values can range from primitive data types (integers, floats, strings, etc.) to more advanced data structures such as, lists and hash-tables, in some key-value stores, like Redis.

In many ways, key-value stores can be thought of as distributed dictionaries rather than a relational database. For example, in the context of social media site Instagram, they use Redis to store a mapping between media ID (keys) and user ID (values) to the order of 300 million key-value pairs [1].

Key-value stores can reside in a servers main memory or on disk. Key-value stores that reside on a server's main memory are referred to as *in-memory* key value stores. Examples of these systems are memcached and Redis. On the other hand, some key-value stores are persistent, like RocksDB, and write to disk occasionally. In addition, there are key-value stores designed specifically for non-volatile memory (NVM) such as `pmemkv`, developed by Intel. Our thesis will focus on in-memory key-value stores.

2.2 Multi-Tenancy

Figure 1.1 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. Multi-tenant environments are attractive for data centers since they allow resources to be shared among applications. In fact, in his thesis, Denning showed that sharing resources leads to more efficient utilization [19].

Prior work done in the multi-tenant environment has been centered around virtual machine management [47]. One physical server may host several virtual machines, each with variable memory demands. The work by Weiming and Wang [47] introduces a memory balancer that allocates memory among virtual machines on a physical host. They show that the memory balancer provides an increase in overall system throughput. Further motivating the multi-tenant architecture for the data center environment.

2.2.1 Sharing Models

The key component is resource sharing is how to decide *how much* to give each application. From the literature and in practice,there have been several different notions of how to share the memory. We call these *sharing models* for application memory and we present three different allocation techniques in order of complexity.

2.2.2 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 [5, 34]. 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 manual tuning by system administrator.

2.2.3 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 [17].

2.2.4 Working-Set Size Allocation

Formally, researchers [2, 43] defined the multi-tenant allocation problem as the following optimization problem: compute the ideal memory $\mathbf{m} = [m_1, m_2, ..., m_N]$ that minimizes the number of misses in the cache,

$$\begin{aligned} &\text{minimize} \quad F(\mathbf{m}) = \sum_{i=1}^{N} mrc_i(m_i) * NR_i \\ &\text{subject to} \quad \sum_{i=1}^{N} m_i \leq M \end{aligned} \tag{2.1}$$

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 misses is $mrc_i(m_i) * NR_i$.

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 [43].

2.3 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.*

2.3.1 Average Eviction Time (AET)

Hu et al. present a series of kinetic equations related to average data eviction in the cache [27, 28]. 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 [23, 49], which counts the number of distinct accesses between an access and its next reuse. Table 2.1 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:

$$\int_0^{AET(c)} P(t)dt = c \tag{2.2}$$

where $P(t)$ is the probability that a reference has reuse time greater than t. We

Table 2.1
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

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.:

$$mrc(c) = P(AET(c)) \tag{2.3}$$

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.

2.3.2 Cliffhanger

In contrast to computing the full miss ratio curve, another approach is to estimate the working set size by using a data structure that tracks recently evicted keys, called *shadow queues*. Shadow queues are similar to victim caches [29], except that they only store the *keys* of the evicted items. The Cliffhanger [16] miss ratio curve estimation technique uses these shadow queues to hold the keys of recently evicted objects, in order tell how often data that was *recently* evicted (that is, residing in the shadow queue) is reused.

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. One credit is a pre-defined amount of memory that is taken from a victim application at random.

Cliffhanger is the sharing method that is implemented in the recent multi-tenant key-value store Memshare by Cidon et al [17]. 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. The victim application is chosen at random. Memshare employs log structured memory, so applications are cleaned based on their *need*. An application's need is defined as the amount of memory *in use* over the amount of memory *currently allocated*. A need of less than 1 means the application is currently over-provisioned and the Memshare system will prefer to clean items from that particular application. When items are cleaned, they are pushed into the application's shadow queue. Therefore allowing the application to receive more credits (space) if those items are reused soon.

Chapter 3

mPart

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 2.2 and 2.3. 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

We present a dynamic programming optimization algorithm. We choose this method in order to achieve polynomial time in determining memory assignments since the problem can have an exponential search space.

3.2.1 Dynamic Programming

At a specified interval, we run the arbitration algorithm presented in [43]. 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 amount memory size j. We use k to denote the amount of memory for application i. 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:

$$\begin{aligned} &miss(i,j) = \\ &\min(miss(i,j), miss(i-1, j-k) + mrc_i(k) * NR_i) \end{aligned} \tag{3.1}$$

where $L_i \leq k \leq H_i$, since k represents the amount of memory that we will assign to application 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:

$$miss(1,j) = mrc_i(j) * NR_i \tag{3.2}$$

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 $O(N^2 \times (\frac{M}{S})^2)$.

Algorithm 1 Dynamic Programming Arbitration.

```
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}
 1: for $i \in V$ do
 2:     $E_i \leftarrow \max(low_i, WSS_i)$
 3: end for
 4: $E_s \leftarrow \sum E_i$
 5: if $M \geq E_s$ then
 6:     $bonus \leftarrow M - E_i$
 7:     for $i \in V$ do
 8:         $A_i \leftarrow bonus \times \frac{E_i}{E_s}$
 9:     end for
10:
11:     return {$A$}
12: else
13:     $Low_s \leftarrow 0$, $High_s \leftarrow 0$ {bound on inner $j$ loop}
14:     for $i \in V$ do
15:         $Low_s \leftarrow Low_s + L_i$
16:         $High_s \leftarrow High_s + H_i$
17:         for $j \leftarrow Low_s; j \leq \min(M, High_s)$ do
18:             if $i = first$ then
19:                 $miss[1][j] \leftarrow mrc_i(k) * NR_i$
20:             else
21:                 $miss[i][j] \leftarrow \infty$
22:                 for $k \leftarrow L_i; k \leq H_i$ do
23:                     if $j \geq k$ then
24:                         $cMiss \leftarrow mrc_i(k) \times NR_i$
25:                         $pMiss \leftarrow miss[i-1][j-k]$
26:                         $nMiss \leftarrow pMiss + cMiss$
27:                         if $miss[i][j] \geq nMiss$ then
28:                             $miss[i][j] \leftarrow nMiss$
29:                             $Target[i][j] \leftarrow k$
30:                         end if
31:                     end if
32:                     $k \leftarrow k + S$
33:                 end for
34:             end if
35:         end for
36:     end for
37:     $T \leftarrow M$
38:     for $i \leftarrow N; i \rightarrow 1$ do
39:         $A_i \leftarrow Target[i][T]$
40:         $T \leftarrow T - Target[i][T]$
41:     end for
42:
43:     return {$A$}
44: end if
```

Chapter 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 [11] and varying number of objects (ETC, PSA, YCSB) from [14, 18, 25]. Table 4.1 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 [17] to measure hit ratios and

Table 4.1
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 4.2
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

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 4.1 and vary the initial allocations. We set the following allocations:

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

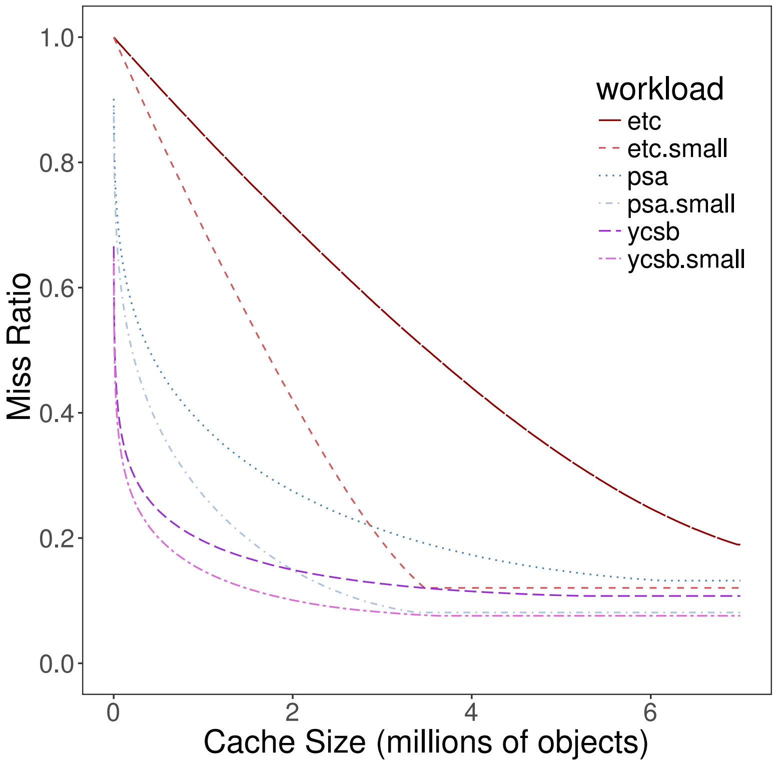


Figure 4.1: Miss Ratio Curve for each application used in our experiment.

set size. We set the total memory to be 90% of the sum of all applications working set size.

Table 4.2 compares each allocation and Figure 4.1 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 an 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 4.2 shows the hit ratio and memory allocations of each application over the number of accesses.

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

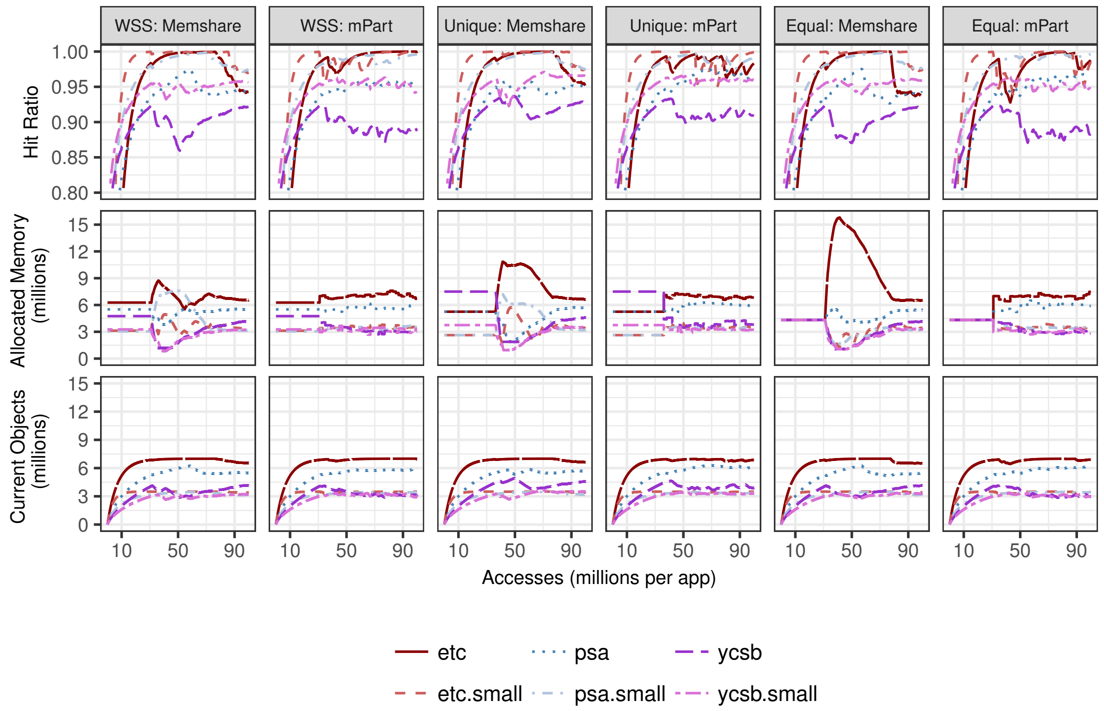


Figure 4.2: Hit ratio and Memory Allocation vs. Number of Accesses for each application

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 4.2 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 4.3 summarizes the miss ratios 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.

4.1.1 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.

4.1.2 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, while on the other hand, *etc* is underallocated and hits very often in its shadow queue, causing it to be assigned more credits.

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*, *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.

Table 4.3
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**

4.1.3 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 (its need is less than 1). 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 (89.6% average hit ratio with cache size of 4.1 million to 89.4% 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 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. [5].

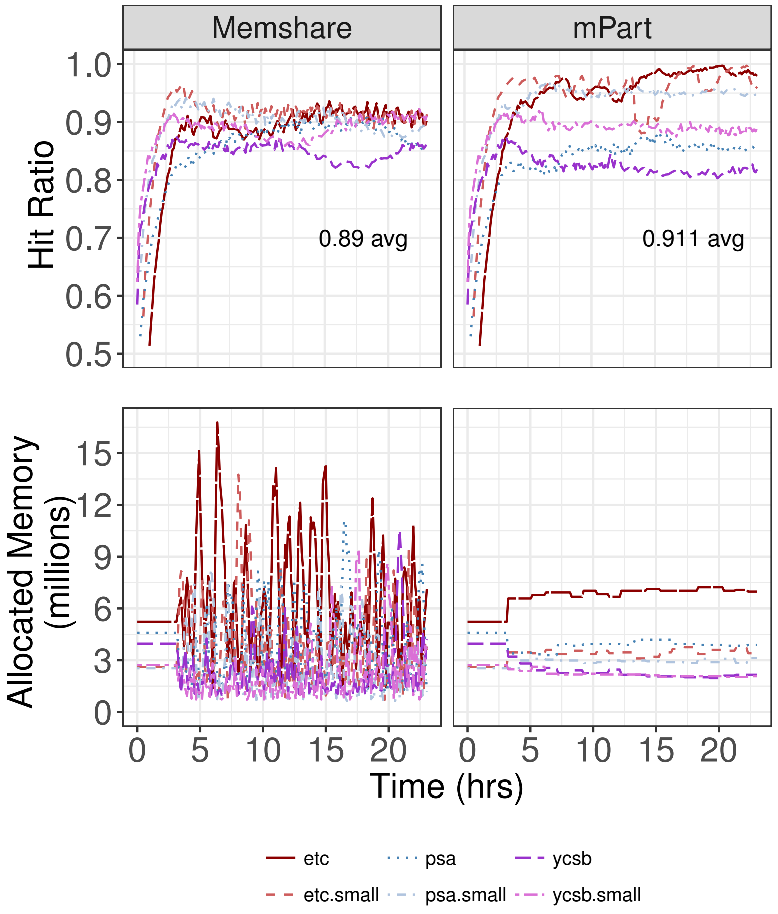


Figure 4.3: Time-Varying Workload over 24 hour period.

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 4.3 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 ratio in the shadow queue changes with the request rate, which in turn, corresponds to the number of credits that are assigned to an application. In Figure 4.3, 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 ratio 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 data set of 100 million objects. The noisy neighbor enters the system at hour 5. The results from both Memshare and mPart are shown in Figure 4.4.

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 4.5 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 [10].

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. 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 4.6. It provides accurate results at a sampling rate at 1/10,000 requests.

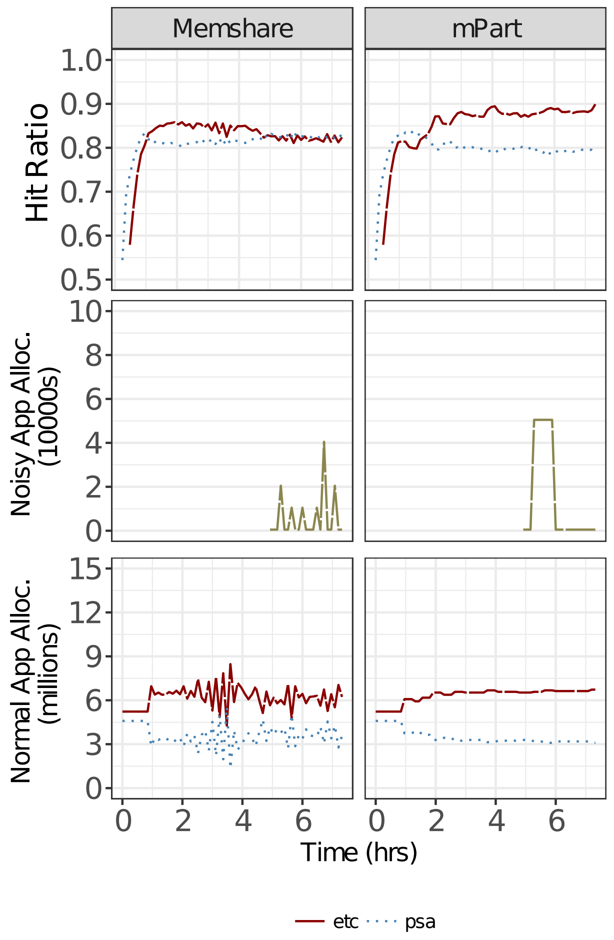


Figure 4.4: Applications with a noisy neighbor added.

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 ($\alpha = 1$). The results are the average of 10 runs and are shown in Tables 4.4 and 4.5.

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 result show a decrease in GET request tail latency since we only need to perform sampling on the critical path of a request. In our experiments, the steady state hit

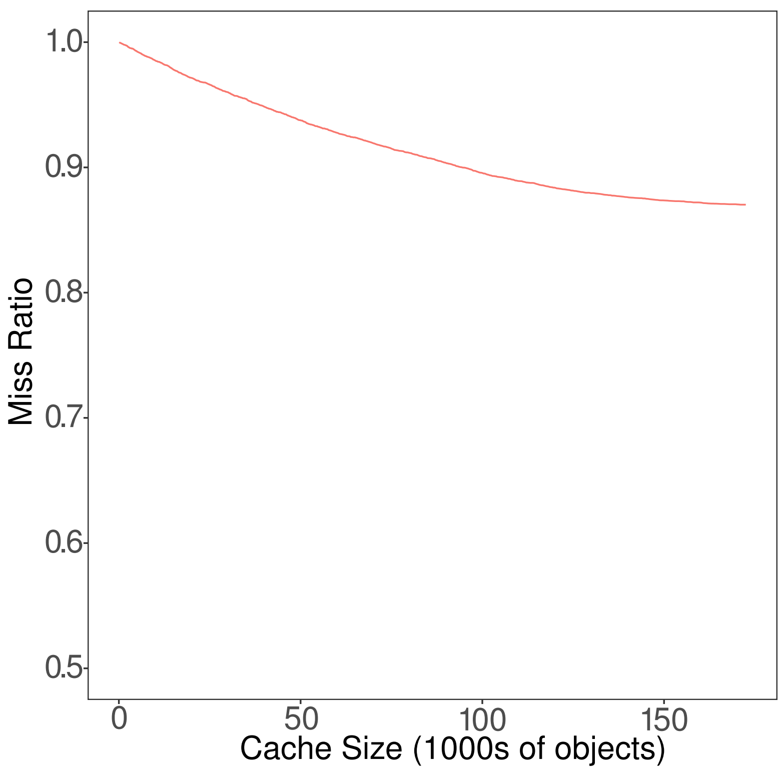


Figure 4.5: Noisy Neighbor's Miss Ratio Curve.

ratios are typically 95% and above, therefore in Memshare, the 95th percentile of GETs reflects the cost of accesses the shadow queue as shown in Table 4.4. On the other hand, in mPart we only sample 1/10,000 of the requests to add to the reuse time histogram. This avoids impacting the tail latency as much as Memshare does.

After a miss in the cache, both systems are configured to issue a SET request. On a SET request we are adding new data to the cache and could cause the log cleaner thread to run. When the log cleaner thread runs in Memshare, applications are sorted by *need* and then items are evicted in LRU order. The evicted items are then pushed to the shadow queue. Again for the same reasons as in the GET request percentiles, this has an effect on the 95th percentile and above latencies. In mPart, when the log cleaner thread runs, there is no shadow queue to push items into.

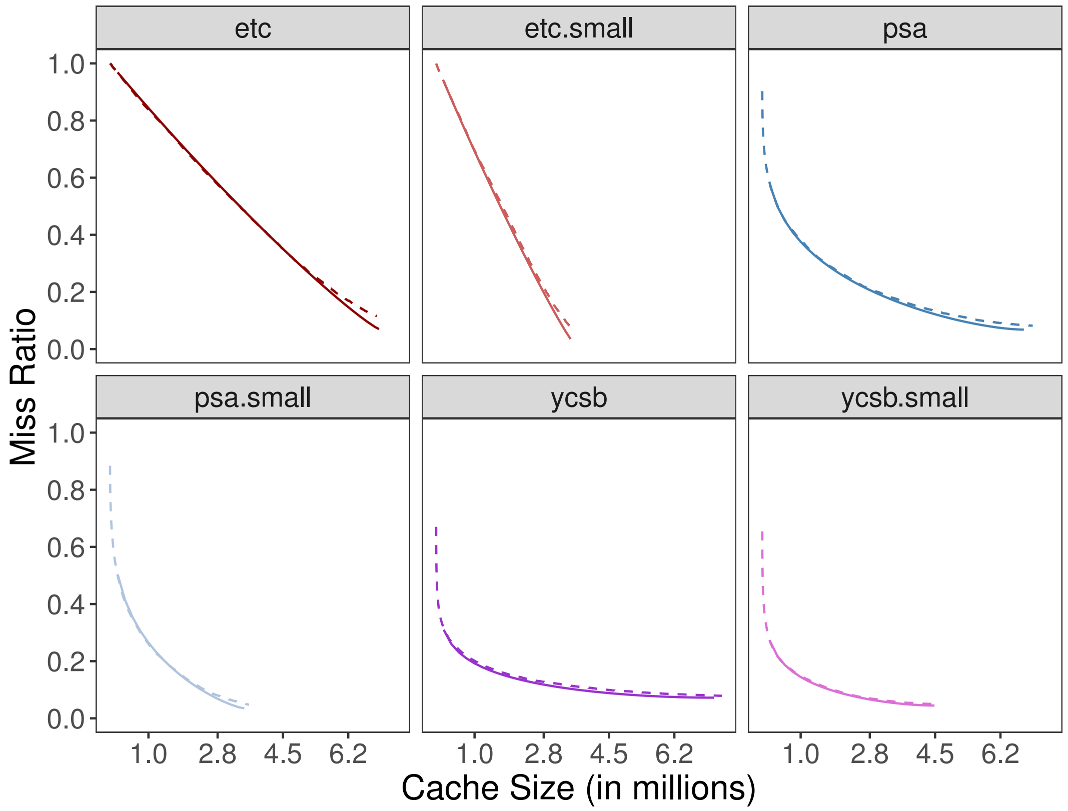


Figure 4.6: Accuracy of MRC generated by AET vs actual.

Table 4.4
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**

4.6 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 4.6 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%

Table 4.5
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 4.6
Percentage of CPU time spent on cleaning and online WSS tracking.

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

Table 4.7
Throughput (1000 op/s) for Memshare and mPart.

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

of CPU time is spent testing shadow queues on GET misses in Memshare.

Table 4.7 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.

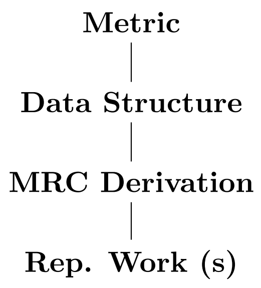
Chapter 5

Related Work

In order to organize the related work, we present a taxonomy of miss ratio curve construction techniques, along with some previous applications of these models.

5.1 Taxonomy of MRC Constructions

Figure 5.1 and 5.2 classify the techniques that use stack distance and reuse time respectively, by how they store and process the data into a miss ratio curve. We also give a representative work from each class. The figures are organized in the following structure from top to bottom; metric (reuse time or stack distance), primary data structure used, miss ratio curve derivation method, and representative works.



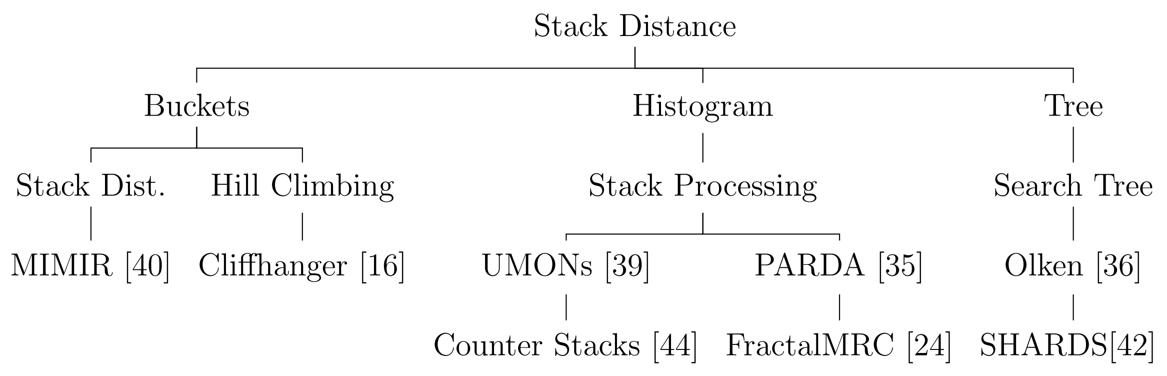


Figure 5.1: Methods based on Stack Distance.

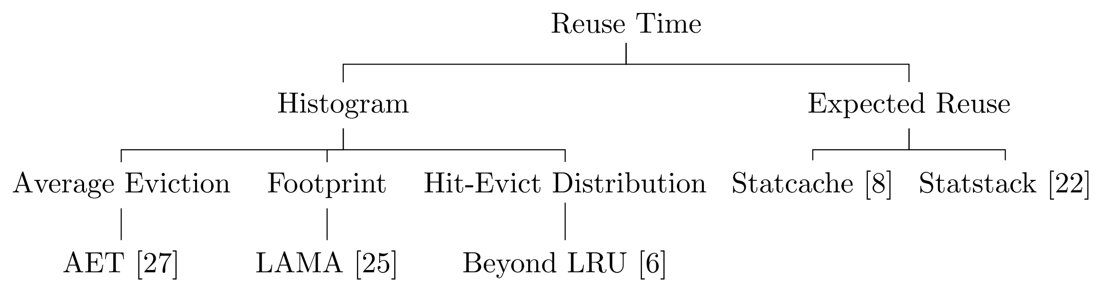


Figure 5.2: Methods based on Reuse Time.

5.1.1 Stack Distance Algorithms

This class of algorithms for calculating miss ratio curves takes advantage of the least recently used (LRU) replacement policy which maintains the stack order. Table 5.1, demonstrates this for a set of requests.

To generate a miss ratio curve for cache size d, a *histogram* of stack distances is created. The stack distance histogram represents a distribution of the application's reuse distance. Given the stack distance histogram, one can calculate the number of hits in a cache of size d as the sum of all the references with stack distances $\leq d$ and similarly the number of misses as the sum of all references with stack distance $> d$.

Formally, let the cache size be d items, N is the total number of requests, and M is the total number of unique items, then the $miss\,ratio(d)$ is express as:

Table 5.1
Example stack distance calculation for a set of requests.

request	**time**	**stack contents**	**stack dist.**
a	1	a	∞
b	2	b,a	∞
c	3	c,b,a	∞
d	4	d,c,b,a	∞
a	5	a,d,c,b	4 (d,c,b)
a	6	a,d,c,b	1 (none)
d	7	d,a,c,b	2 (a)
b	8	b,d,a,c	4 (d,a,c)

$$1 - \underbrace{\frac{\sum_{i=1}^{d} freq(i)}{N}}_{\text{hit ratio}} = \underbrace{\frac{\sum_{i=d+1}^{M} freq(i)}{N}}_{\text{miss ratio}} \tag{5.1}$$

The left hand side of equation 5.1 calculates the miss ratio by first calculating the number of hits in the cache. This is done by counting every item with reuse distance less than or equal to cache size d. Since $misses = 1 - hits$, we arrive at the miss ratio after dividing by the total number of requests.

The right hand side of the equation calculates the miss ratio by first calculating the number of misses in the cache. That is, the number of items with reuse distance greater than cache size d, since any requests for items with reuse distance larger than the cache size will result in a miss as the item would already have been evicted. Dividing by the total number of requests, we arrive at the miss ratio.

It is important to note that some represent this as an integration of the probability density function given by the distribution. The reason is to highlight the order relationship between stack distance and miss ratio, that is, miss ratio is one order higher than stack distance. This result was proved in Ding's Higher Order Theory of Locality (HOTL) [21].

Mattson's Stack Distance Algorithm: Mattson et al. [32] developed the first algorithm to use stack distance as a method for calculating the miss ratio for a given cache size. It has five main steps for each reference x:

1. Search stack to find the current location of x (if any)

2. Calculate the distance d from the top of the stack
3. Update cache miss counters for all sizes $> d$
4. Update cache hit counters for all sizes $\leq d$
5. Push x onto the stack and remove the last location of x (if any)

The algorithm runs in $O(N * M)$ time, where N is total number of references and M is unique number of references. Many works improve on this algorithm by using more efficient data structures, sampling, and estimated stack distances.

Fang et al. [23] showed that the stack distance distribution of a program's typical workload could be predicted accurately given a significantly smaller input of the workload. They predicted the stack distance distribution for the *reference* input set of the SPEC2K benchmark using only the *train* and *test* input sets. These results encouraged researchers to move towards sampling the references in order to build the distribution while still achieving accurate stack distance distributions.

Tree Based: In tree-based stack distance calculations we exploit the logarithmic search time in balanced tree structure. Methods in this class have developed towards approximation of distances by getting to a leaf node that represents an approximate stack distance.

Partial Sums Tree: by Bennet et al. [7] introduce a partial sums tree based on binary vectors representing the stack distance of a reference. First, they introduce the concept of using a hash table to store the *previous* location p of a reference. Second, they maintain a binary vector, B, at time t for a given reference x with the following property:

$$B_t = \begin{cases} 1 & \text{if } x_p \neq x_i (i = p, p+1, ..., t) \\ 0 & \text{otherwise.} \end{cases}$$

Now, the number of 1s in B_t is the stack distance of reference x. As t increases the idea is to create a hierarchy of partial sums out of B. For example, imagine that every 3 time intervals all the 1s are summed and stored as a partial sum B^i. Then for every 3 partial sums B^i, there is a new final sum. In order to, calculate the stack distance now, we do a tree traversal from our current time t to the previous use time p. Since each lookup in the hash table is $O(1)$ and each stack distance calculation is $O(log(n))$, then for a trace of N references the running time is now $O(N * log(n))$.

Interval Tree: by Almasi et al. [3] interpret Bennet's binary vectors in a different manner. They use only the 0s from the vector, calling them holes. The calculation for stack distance now becomes:

$$stackdist(x) = t - p - holes(x_p)$$

Where t is current time, p is previous time, and $holes(x_p)$ represents the number of holes (0s) between t and p in B. Now let a hole's index be h_i and let the interval $[h_i, h_j]$ represent the time between two consecutive holes.

Since these intervals of holes are non-overlapping, they can be represented as AVL or red-black trees. For N references this takes $O(N * log(n))$, similar to Bennet's asymptotic bound, but in practice it performs significantly better because the tree is well balanced. This algorithm uses the same space cost as Bennet's algorithm, $O(M)$ where M is the number of accesses in the trace.

Search Tree: Olken proposed ordering the LRU stack as a binary tree based on access times so that an in-order traversal gives the LRU stack position [36] . For a given reference at the root, the right side of the tree represents references used more recently and the left side represents older references. At each node in the tree, they store a count of the number of nodes in the right subtree (more recent) and the number of nodes in the left subtree (older).

Now in order to calculate a stack distance, on a reference to x, one searches the tree for its location and traverse to the top of the tree, recording the number of nodes in the right subtree of each node that they visited. The algorithm runs in $O(N * log(n))$ time and needs to store M number of nodes, where M is the number of unique accesses.

Scale Tree: In Zhong et al. [49] they produce a modified version of Olken's search tree by modifying it to support a *range* of times at each node. This maps several references to one node and allows for much faster searches. The amount of error incurred by using a range of times at each node is bounded by the range. The overall run time is reduced to $O(N * log(log(M))$, but the algorithm still require $O(M)$ space.

SHARDS: Spatially hashed reuse distances (stack distances), or SHARDS, is a recent advancement to constant space use by Waldspurger et al. [42]. They achieve this by sampling accesses based on the hashed value of their location, hence spatially hashed. These samples are then put into an interval tree to compute their stack distance. After computing stack distance, it is recorded in the stack distance histogram in order to solve for the miss ratio curve. By using fixed size hash table and fixed size LRU tree, each reuse distance sample takes constant, $O(1)$ space. Allowing for efficient construction of the stack distance histogram.

Buckets: These algorithms employ bins to reduce the number of stack positions that exist in the LRU stack. The key insight is to reduce n stack positions into m buckets while still maintaining the LRU stack order.

MIMIR: by Saemundsson et al. [40] creates a fixed number of buckets to store references. Each bucket B maintained the LRU stack order property, that is B_i has lower stack distance than B_j since $i < j$. While buckets can vary in size, they introduce an *aging* procedure to ensure that buckets stay balanced. A common metric used to resize the buckets is the *average* stack distance.

The miss ratio curve is generated from the estimated stack distance distribution. For any bucket the actual stack distance lies somewhere between the sum of the size of the buckets before B_i, call it n and $n + sizeof(B_i)$. The stack distance histogram is created from the intervals recorded and the integration over the histogram is done to get the miss ratio at for a cache size (stack distance) d.

Histogram: These methods either use: (1) a set of counters to represent stack distances at each cache size or (2) a set of counters to represent *total* accesses and *unique* accesses. The counts then represent a histogram of the stack distances that are used to construct the miss ratio curve.

Powers of Two: This method by Kim et al. [30] changes Mattson's stack processing algorithm by storing stack distances in discrete intervals of powers of 2. The idea is based on the insight that memory designers often do not require miss ratios for continuous memory sizes but powers of 2 (2, 4, 6 ...) are sufficient. The algorithm stores the approximated stack distances in a linked list of data blocks, which correspond to the LRU *stack* and a hash table is used to find the location of references in the stack. On a reference to a block, the approximated stack distance (in powers of 2) is updated only when a block moves into the next interval.

The algorithm runs in $O(N * M)$ where N is the number of requests and M is the number of intervals that stack distances can fall into. However, since the stack distance is only updated when a block moves into the next interval, the number of expected stack distance updates is the sum of the miss ratios at each interval.

This algorithm has been applied to build miss ratio curves in Zhou et al. [50]. They present a hardware and software implementation for calculating page miss ratio curves to guide memory management in a multi-program environment.

At the hardware level, Tam et al. [41] have used this algorithm to manage L2 cache partitioning. Here they create an access trace to L2 by recording the misses of L1,

Table 5.2
Example Counter Stacks calculation.

a	**b**	**c**	**a**
1	2	3	3
	1	2	3
		1	2
			1

then they send the access trace log into the algorithm given by Kim. Their miss ratio curve is then used to guide how much of the L2 cache should an application receive.

CounterStacks: This recent development by Wires et al. [44] uses probabilistic counters (HyperLogLogs) and Bloom filters (for set inclusion) in order to maintain a list of unique accesses over a period of time. Specifically, they construct a list at time t that contains the number of unique accesses. This allows us to keep track of how much a counter has increased (stack distance) over a number of non-distinct accesses. An example is given in Table 5.2.

In order to derive the stack distance, the *intra-counter* change is calculated. In order to find the last reference, the algorithm checks the newest update to the counter. In Table 5.2 we find the last reference to a is at position 1, hence the stack distance of a lies at position (1,4) = 3. Note that since these counters are probabilistic (HyperLogLogs), their memory usage is logarithmic to the number of distinct items. Therefore, CounterStacks achieves logarithmic space complexity, $O(log(M))$ with its probabilistic counters which set a small bound on the overall error in stack distance calculation.

UMONs: Qureshi et al. [39] developed utility monitor circuits (UMONs) that approximate hit ratio vs. number of ways in an LRU cache. They perform way-partitioning based off the approximated hit ratio curves. UMONs use access counters to record the number of misses at each stack position for a number of ways. Since the cache obeys the LRU policy, the hit ratio for an n-way partition is found by summing the counters up to n-ways.

This process is used in partitioning caches by way per application. If application A has a higher utility for a number of ways (i.e. its hit ratio curve is not as steep) then it will be assigned that number of ways. Application B will be left with the remaining number of ways.

Fractals: L. He et al. [24] use a fractal equation to calculate the miss ratio of a given cache size from the time between two consecutive misses. This is known as the inter-miss gap described by Denning and related to LRU miss ratio by Ding in HOTL [21]. From the distribution of the inter-miss gaps, the LRU miss ratio for a given cache size c, is derived by :

$$intramiss(c) = \frac{1}{missratio(c)}$$

FractalMRC is implemented using hardware performance counters and their model allows for online processing at the cost of a 2% slowdown. One drawback is that their model ignores cold misses which leads to inaccuracy (the cold miss equation is not fractal). On average in the SPEC2006 benchmark they report 76% percent miss ratio curve accuracy.

PARDA: Q. Niu [35] et al. introduce the first explicit parallel stack distance algorithm based on Mattson's stack processing. PARDA exploits the independence of the *define-use* chains in references. That is, only the reference to the last location matters; all previous references are independent. Therefore, one can divide the trace into chunks that contain all references between the *last use* and the *current use* for a given access. The algorithm is outlined below:

1. Start at the end of the trace with reference x
2. Scan until there is another reference to x
3. Send that interval off for processing via Mattson's algorithm
4. Repeat with the next reference after x

5.1.2 Reuse Time Algorithms

More recently, reuse time has emerged as a metric used to approximate miss ratio curves. In 1968, Denning proposed that the reuse time distribution can yield the working set size at time t, but until recently it was hard to achieve accurate results with the analytical models available.

Histogram Based: These methods bin the data recorded into a histogram, and build a miss ratio curve from that histogram by integrating the probability density function or summing the total frequencies over a given range.

Footprint: Xiang et al. [45] introduce the *footprint* function as a mapping from an execution window w to the expected amount of data accessed during w. The key metrics that are measured are: reuse time distribution, first use of each reference, last use of each reference.

Since there can be exponential number of execution windows, The *average* footprint is measured for a set of references. Given the footprint of a set of references, the miss ratio for a cache size c is found by taking the difference between two consecutive execution window sizes.

The footprint algorithm runs in $O(N)$ and $O(M)$ where N is the total number of accesses and M is the total unique accesses. The algorithm was recently applied in LAMA (Locality Aware Memory Allocation) [25], to guide memcached's slab reallocation problem.

Hit-Evict Distribution: Beckmann et al. [6] introduces two new histograms of information, hit and evict. The hit distribution is the frequency of hits for given reuse times. Likewise, the evict distribution is the frequency of evicts for given reuse times. The motivation behind this work is to reduce the dependence on the LRU stack property that many analytic models assume. In their work they also create an *age* distribution based off of the reuse times. This age distribution when combined with the hit and evict distribution can give an accurate LRU model for cache miss ratio.

Statcache: Statcache by Berg et al. [8] uses random sampling to select some memory references for their analysis. For each memory reference sampled it does the following:

1. Set a (hardware) watchpoint on the address and record current number of memory accesses at that time n.
2. On the watchpoint trap read the new current number of memory access N. The reuse time of this address is now $N - n$.
3. Update the reuse time distribution with the new reuse time.

The Statcache model assumes a cache with random replacement, therefore the probability that a cache line is still in a cache after a cache miss is uniform. Since the model calculates the probability that a reference will cause a cache miss, and by assuming miss ratio does not change over time, we can fix miss ratio and derive the probability that a number of references cause a miss from its reuse distance from their reuse time.

Statstack: The Stackstack [22] model builds off of the Statcache work, this time 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.

5.2 MRC Based Allocation

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 [25, 26] achieves this by using the *footprint* metric introduced by Ding et al [21]. MIMIR [40] and Dynacache [15] also approach the problem by modeling the miss ratio for each slab class.

In the context of virtualization, memory allocation between virtual machines on a single host has been studied extensively [48]. mPart uses the dynamic programming algorithm presented by Wang et al. [43] 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.

Chapter 6

Conclusion and Future Work

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.

There are multiple lines for future work. The first is to work on cache allocation when objects are non-uniform size. This problem has been shown to be NP-hard and recent work has given upper and lower bounds on optimal replacement with variable sizes [9]. In a sense, providing a miss ratio curve for each *size class*, as done in LAMA [25], provides a first step, but an open problem is how to combine these curves in order to give an overall application miss ratio curve. It may be the case that we do not need to combine the curves and only do allocations at the size class level where we can assume objects are close enough in size.

Second, is exploring the role that MRCs can play in reducing fragmentation at the memory allocator (`malloc`) level. For example, the `jemalloc` allocator needs to assign applications arenas of slabs, each corresponding to a size class. Miss ratio curves can quantify how an application could benefit from additional arenas during runtime. They could also help quantify how many arenas can be reclaimed under changing workloads by analyzing the change in the MRC.

Third, in the context of key-value stores we need to know when to perform re-allocation. By analyzing intra-MRC change we can decide when an application's working set has changed significantly and if it should receive more or less space as a result. Further, there are different methods for solving the optimization problem. If MRCs are convex, then we can use the the allocation algorithm proposed by Denning in his PhD thesis [19].

Finally, in the context of performance, data centers typically scale out horizontally by adding more caching nodes. Additional caching nodes are costly but may be worth the performance gains given an application's miss ratio curve. The benefits of adding/removing caching nodes in a data center can be calculated from the application's miss ratio curve.

References

[1] Instagram engineering team. `https://instagram-engineering.tumblr.com/post/12202313862`. Accessed: 2016-12-10.

[2] Abad, C. L., Abad, A. G., and Lucio, L. E. Dynamic memory partitioning for cloud caches with heterogeneous backends. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (New York, NY, USA, 2017), ICPE '17, ACM, pp. 87–90.

[3] Almási, G., Caşcaval, C., and Padua, D. A. Calculating stack distances efficiently. *SIGPLAN Not. 38*, 2 supplement (June 2002), 37–43.

[4] Amazon. Amazon web services, May 2018.

[5] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. 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* (New York, NY, USA, 2012), SIGMETRICS '12, ACM, pp. 53–64.

[6] Beckmann, N., and Sanchez, D. Modeling cache performance beyond lru. In *High Performance Computer Architecture (HPCA), 2016 IEEE International Symposium on* (2016), IEEE, pp. 225–236.

[7] Bennett, B. T., and Kruskal, V. J. Lru stack processing. *IBM Journal of Research and Development 19*, 4 (1975), 353–357.

[8] Berg, E., and Hagersten, E. 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)* (2004), pp. 20–27.

[9] Berger, D. S., Beckmann, N., and Harchol-Balter, M. Practical bounds on optimal caching with variable object sizes, June 2018.

[10] Blankstein, A., Sen, S., and Freedman, M. J. Hyperbolic caching: Flexible caching for web applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 499–511.

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

[12] Byrne, D., Onder, N., and Wang, Z. mpart: Miss-ratio curve guided partitioning in key-value stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2018), ISMM 2018, ACM, pp. 84–95.

[13] Carra, D. Benchmarks for testing memcached memory management, December 2016.

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

[15] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (Santa Clara, CA, 2015), USENIX Association.

[16] Cidon, A., Eisenman, A., Alizadeh, M., and Katti, S. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 379–392.

[17] Cidon, A., Rushton, D., Rumble, S. M., and Stutsman, R. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)* (Santa Clara, CA, 2017), USENIX Association, pp. 321–334.

[18] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (New York, NY, USA, 2010), SoCC '10, ACM, pp. 143–154.

[19] Denning, P. J. *Resource allocation in multiprocess computer systems.* PhD thesis, Massachusetts Institute of Technology, 1968.

[20] Denning, P. J. The working set model for program behavior. *Commun. ACM 11*, 5 (May 1968), 323–333.

[21] Ding, C., and Xiang, X. A higher order theory of locality. In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness* (New York, NY, USA, 2012), MSPC '12, ACM, pp. 68–69.

[22] Eklov, D., and Hagersten, E. Statstack: Efficient modeling of lru caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS '10)* (March 2010), pp. 55–65.

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

[24] He, L., Yu, Z., and Jin, H. Fractalmrc: Online cache miss rate curve prediction on commodity systems. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (May 2012), pp. 1341–1351.

[25] Hu, X., Wang, X., Li, Y., Zhou, L., Luo, Y., Ding, C., Jiang, S., and Wang, Z. LAMA: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, 2015), USENIX Association, pp. 57–69.

[26] Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., Jiang, S., and Wang, Z. Optimizing locality-aware memory management of key-value caches. *IEEE Transactions on Computers (TC '17) 66*, 5 (May 2017), 862–875.

[27] Hu, X., Wang, X., Zhou, L., Luo, Y., Ding, C., and Wang, Z. Kinetic modeling of data eviction in cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)* (Denver, CO, 2016), USENIX Association, pp. 351–364.

[28] Hu, X., Wang, X., Zhou, L., Luo, Y., Wang, Z., Ding, C., and Ye, C. Fast miss ratio curve modeling for storage cache. *ACM Trans. Storage (TOS'18) 14*, 2 (Apr. 2018), 12:1–12:34.

[29] Jouppi, N. P. 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)* (May 1990), pp. 364–373.

[30] Kim, Y. H., Hill, M. D., and Wood, D. A. Implementing stack simulation for highly-associative memories. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems* (New York, NY, USA, 1991), SIGMETRICS '91, ACM, pp. 212–213.

[31] Labs, R. redis, May 2018.

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

[33] memcached. memcached, 2018.

[34] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)* (Lombard, IL, 2013), USENIX, pp. 385–398.

[35] Niu, Q., Dinan, J., Lu, Q., and Sadayappan, P. Parda: A fast parallel reuse distance analysis algorithm. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium* (May 2012), pp. 1284–1294.

[36] Olken, F. Efficient methods for calculating the success function of fixed-space replacement policies. Tech. rep., Lawrence Berkeley Lab., CA (USA), 1981.

[37] Ou, J., Patton, M., Moore, M. D., Xu, Y., and Jiang, S. A penalty aware memory allocation scheme for key-value cache. In *2015 44th International Conference on Parallel Processing* (Sept 2015), pp. 530–539.

[38] Pu, Q., Li, H., Zaharia, M., Ghodsi, A., and Stoica, I. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)* (Santa Clara, CA, 2016), USENIX Association, pp. 393–406.

[39] Qureshi, M. K., and Patt, Y. N. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (Washington, DC, USA, 2006), MICRO 39, IEEE Computer Society, pp. 423–432.

[40] Saemundsson, T., Bjornsson, H., Chockler, G., and Vigfusson, Y. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, ACM, pp. 28:1–28:14.

[41] Tam, D. K., Azimi, R., Soares, L. B., and Stumm, M. Rapidmrc: Approximating l2 miss rate curves on commodity systems for online optimizations. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2009), ASPLOS XIV, ACM, pp. 121–132.

[42] Waldspurger, C. A., Park, N., Garthwaite, A., and Ahmad, I. Efficient mrc construction with shards. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies* (Berkeley, CA, USA, 2015), FAST'15, USENIX Association, pp. 95–110.

[43] Wang, Z., Wang, X., Hou, F., Luo, Y., and Wang, Z. Dynamic memory balancing for virtualization. *ACM Trans. Archit. Code Optim. (TACO '16) 13*, 1 (Mar. 2016), 2:1–2:25.

[44] Wires, J., Ingram, S., Drudi, Z., Harvey, N. J. A., and Warfield, A. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (Broomfield, CO, 2014), USENIX Association, pp. 335–349.

[45] Xiang, X., Bao, B., Ding, C., and Gao, Y. Linear-time modeling of program working set in shared cache. In *2011 International Conference on Parallel Architectures and Compilation Techniques* (Oct 2011), pp. 350–360.

[46] Yang, T., Berger, E. D., Kaplan, S. F., and Moss, J. E. B. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2006), OSDI '06, USENIX Association, pp. 103–116.

[47] Zhao, W., and Wang, Z. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 21–30.

[48] Zhao, W., and Wang, Z. Dynamic memory balancing for virtual machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2009), VEE '09, ACM, pp. 21–30.

[49] Zhong, Y., Shen, X., and Ding, C. Program locality analysis using reuse distance. *ACM Trans. Program. Lang. Syst. (TOPLAS '09) 31*, 6 (Aug. 2009), 20:1–20:39.

[50] Zhou, P., Pandey, V., Sundaresan, J., Raghuraman, A., Zhou, Y., and Kumar, S. Dynamic tracking of page miss ratio curve for memory management. In *ACM SIGOPS Operating Systems Review* (2004), vol. 38, ACM, pp. 177–188.