PR-MRC: MRC Construction using Non-Statistical Sampling

by

Albert Lee

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Abstract

PR-MRC: MRC Construction using Non-Statistical Sampling

Albert Lee

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2022

Miss ratio curves (MRCs) play an important role for visualizing and deciding on an effective cache size for a given workload. Unfortunately, generating exact MRCs require processing and memory overheads that are not practical for online use in production systems.

Previous approximate algorithms employ statistical sampling where the set of sampled keys is decided based on satisfying a random sampling condition. We introduce a new approximation algorithm that employs non-statistical sampling where the set of sampled keys is decided based on prior knowledge from previously accessed keys. Our algorithm, called PR-MRC (Pattern Recognition MRC), only updates the MRC when the access pattern is changing.

We evaluate PR-MRC using publicly available traces. More than eighty MRCs were generated and compared to the exact MRC. Results show up to 5.5% lower throughput for MRC generation compared to existing approximation algorithms, while achieving improved MRC accuracy.

Contents

1 Introduction **1**

2 Background & Related Work **6**

2.1 Fixed Sized Data Block Algorithms 7

2.1.1 Mattson 7

2.1.2 Olken 8

2.1.3 SHARDS 9

2.1.4 CounterStacks 10

2.1.5 AET 11

2.1.6 Zhong and Chang 12

2.1.7 Low Cost Working Set Size Tracking 13

2.2 Variable Sized Data Block Algorithms 14

2.3 Measuring Errors in the Generated MRC 17

3 MRC Generation with Non-Statistical Sampling **19**

3.1 Statistical Sampling Issues 20

3.2 Algorithm Overview 22

3.3 Low Level Algorithm Discussion 23

3.3.1 Identifying MRC Convergence 23

3.3.2 Identifying Changes in the Workload Access Pattern 28

4 Evaluation **34**

4.1 Experimental Setup 34

4.2 Configuration Analysis 37

4.3 Accuracy of PR-MRC 40
4.4 Throughput of PR-MRC 47
4.5 Memory Usage of PR-MRC 49

5 Concluding Remarks **53**

Bibliography **56**

List of Tables

3.1 Comparison of Most Frequently Accessed Key for Web Workload 22
3.2 Calculated average absolute miss ratio difference for *prxy* workload used to determine convergence. 25
3.3 Calculated maximum slope difference for *web* workload used to determine convergence 28
3.4 Working Set Size Difference across multiple Twitter Cluster25 workload intervals. . . 30
3.5 Approximate miss rate where *cSize* = 160MB across multiple Twitter Cluster7 workload intervals. 32
3.6 Comparison of the set of the 10-most frequently accessed keys between two adjacent intervals for the Twitter Cluster25 workload. 33

4.1 Workloads used for evaluation. 35
4.2 The thirteen specific Twitter workloads that we run our experiments on. 37
4.3 Comparison of the set of the 10-most frequently accessed keys for Olken and SHARDS R = 0.1 for the `SEC 2017-QRT2` workload. 43
4.4 Average Throughput Comparison between Exact, PR-MRC, Shards R = 0.1, and SHARDS R = 0.01 for MSR, SEC, and Twitter workloads. 48
4.5 Average Throughput (accesses per second) across Twitter traces using different PR-MRC criteria during the Access Pattern Monitoring Phase. 50
4.6 Memory Usage across all MSR, SEC, and Twitter traces using different PR-MRC criteria during the Access Pattern Monitoring Phase. 51

List of Figures

1.1 Miss Rate Curve for the MSR web trace 2

2.1 Example of the operation of the CounterStacks algorithm. 10

2.2 Cumulative distribution function for the different traces from the MSR collection. (Between brackets is the number of block sizes used by the trace 14

2.3 Miss rate curves for different block size treatment for the "proj" workload from MSR traces. 15

2.4 Example of the modified CounterStacks algorithm to support variable block sizes. . . 16

2.5 Comparison of MRCs generated with sampling and without sampling for the *ts* workload. 17

3.1 Comparison of MRCs generated with sampling and without sampling for the *web* workload. 21

3.2 Captured MRCs for *prxy* workload used in the error convergence method. 24

3.3 Captured MRCs for *web* workload used in the slope convergence method. 26

3.4 Methodology of the *Access Pattern Monitoring Phase*. Each colored box represents an access pattern. 30

4.1 MAE of 12 distinct PR-MRC configurations tested on 13 Twitter traces, including the average MAE for each configuration shown as a red X. Note the logarithmic scale of the y-axis. 38

4.2 Throughput of 12 distinct PR-MRC configurations tested on 13 Twitter traces, including the average throughput for each configuration shown as a red X. 39

4.3 Average MAE vs Average Throughput of 12 distinct PR-MRC configurations tested on 13 Twitter traces. 40

4.4 Average MAE vs Average Throughput using the "best" PR-MRC configuration and varying the interval size I_{MP} on 13 Twitter traces. 41

4.5 Comparison of MRCs generated by Olken, PR-MRC, SHARDS R = 0.1, and SHARDS R = 0.01 for `SEC 2017-QRT2` workload. 42

4.6 Comparison of MRCs generated by Olken, PR-MRC, SHARDS R = 0.1, and SHARDS R = 0.01 for `SEC 206-QRT3` workload. 42

4.7 MAE of PR-MRC against Olken across all tested workloads, including the average MAE for each configuration shown as a red X. Note the logarithmic scale of the y-axis. 43

4.8 Miss Ratio Comparison between 25% of the workload processed and 100% of the workload processed for the `Cluster18` workload using rounded next power of two block sizes. 45

4.9 MRC Comparison between the exact MRC and PR-MRC when using each criteria independently for the `Twitter Cluster44` workload. 45

4.10 MAE of SHARDS R = 0.1 against Olken across all tested workloads, including the average MAE for each configuration shown as a red X. Note the logarithmic scale of the y-axis. 46

4.11 Throughput of PR-MRC across all tested workloads, including the average throughput for each configuration shown as a red X. 47

Chapter 1

Introduction

In-memory, DRAM-based caching systems are used to cache data stored in backend storage systems that use SSDs or HDDs. When a user application requests data, the caching system is checked first. If the data is already stored in the caching system, it can be sent back to the application without having the application access the backend storage system. The read access time of the requested data is reduced because the data obtained from the caching system is served out of memory instead of persistent storage. The load on the persistent storage system may also be reduced because a portion of the read accesses are served from the cache, which potentially reduces the load on the backend storage system and which in turn reduces response times and improves scalability. As such, in-memory caching servers are prevalent in today's distributed systems. Facebook [17], Amazon [3], and Shopify [1] are examples of companies that use variations of Redis [21] and Memcached [8], both popular open-source in-memory caching systems. Both systems store data as key-value pairs, where the data is a value that is retrieved using a unique key.

An important aspect of configuring an in-memory cache is the amount of DRAM that is allocated to the cache, as the size of the memory determines the effectiveness of the cache. If the cache is too small, then objects stored in the cache will have to be evicted more often due to the small size of the cache, leading to a higher miss rate. A higher miss rate renders the cache less effective than it could be, as the miss rate directly affects the data access response time in a significant way; e.g., a 10% increase in miss rate can result in a 300x slower response time or more [13]. If the cache is too large (e.g. larger than the working set size [29] of the application using the cache), then the allocated DRAM is not being used efficiently because the same miss rate can be achieved with a

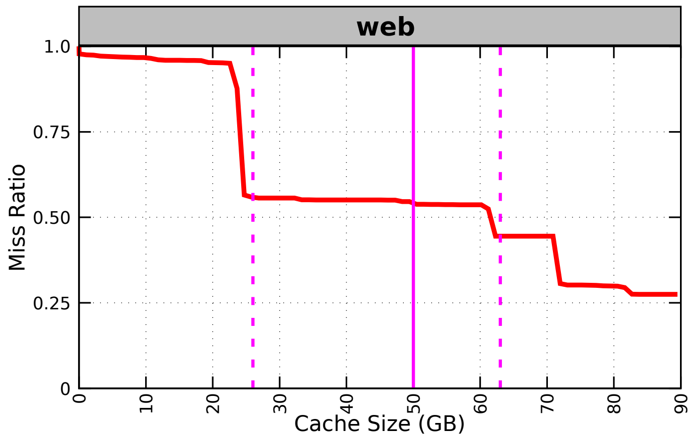

Figure 1.1: Miss Rate Curve for the MSR web trace

smaller DRAM footprint. This is relevant as the price of memory has been steadily increasing over the past few years [11].

Miss rate curves (MRCs) are an important tool to help decide on an effective cache size. MRCs plot the percentage of cache misses as a function of the cache size for a given workload. Thus, an MRC shows how a change to the size of the cache affects the miss rate for a given workload. Figure 1.1 shows an example. The MRC shown in the figure is specific to the "web" workload made available by Cambridge Microsoft Research Lab (MSR) [16]. It shows that if 50GB is allocated to the cache, then the miss ratio is just above 50%. It also shows that the cache can be configured with only 25GB of DRAM at the cost of a minimal increase in the miss rate, or alternatively the cache can be configured with 63GB of DRAM to reduce the miss rate by 10% (relative to the 50GB cache size configuration).

A key issue with generating an MRC is that they require high processing and memory overheads. Mattson was the first, in 1970, to present an algorithm that could generate an MRC using only a single pass over a trace of the target workload's data accesses [15] by maintaining (and updating on each access) a list of accessed keys ordered by access recency. Prior to Mattson's algorithm, MRCs had to be generated by running a separate cache simulation for each cache size of interest. Still, Mattson's algorithm exhibits processing overheads on the order of O(NM) for a workload with N accesses and M uniquely accessed keys. (This was later optimized to O(N log M) by Olken [18].) This overhead is significant because the MRC is workload dependent, and needs to be periodically recreated to reflect changes in the workload. With high compute overheads, the MRC generation may interfere with the processing of the caching service. Moreover, Mattson's algorithm requires O(M) memory, which could otherwise be used to cache additional data.

Over the past 10 years, several new algorithms have been proposed for generating MRCs that are more efficient than Mattson's. These newer algorithms achieve better throughput by generating MRCs *approximately*, rather than generating an exact MRC. The throughput of generating an MRC is defined as the total number of accesses processed by the algorithm per second. While sacrificing accuracy when comparing the approximate MRC to the exact MRC, the processing overheads are reduced relative to Mattson's algorithm. One such algorithm is CounterStacks [28]. Instead of maintaining and updating a list of accessed keys ordered by access recency, CounterStacks maintains periodically instantiated counters recording the number of distinctly accessed keys that have been accessed since the inception of that counter. By doing so, CounterStacks maintains a compact representation of the workload's data accesses that provides enough information to approximately generate an MRC. Another approximate MRC generation algorithm, SHARDS [27], uses Mattson or Olken over a randomly sampled subset of the data accesses, effectively reducing the size of the list of accessed keys used to generate the MRC. Finally, AET [12] mathematically describes a way to generate an MRC using only a reuse time histogram, where the reuse time of an accessed key is the time between its access and its previous access to the same key. By generating the reuse time histogram linearly, generating an MRC is also done in linear time. Chapter 2 describes these algorithms in more detail.

All of the approximate algorithms mentioned above that provide better throughput when generating MRCs compared to Mattson and Olken apply a variation of statistical sampling. Statistical sampling is performed by drawing a set of observations randomly from a population distribution [22]. Based on a set of sampling parameters (e.g. how counters are periodically instantiated in CounterStacks, the sampling rate used for randomly sampling a subset of data accesses in SHARDS, and sampling methods used to generate reuse time histograms in faster than linear time for AET), an MRC is approximated by sampling a subset of the total accesses. The parameters used for sampling do not take into account previous accesses or the frequency of the accessed keys when sampling. Thus, they may sample out accesses that are accessed frequently or periodically. As a result, the algorithms that apply statistical sampling can generate MRCs with poor accuracy if the sampling parameters sample out frequently accessed keys.

This dissertation focuses on applying a *non-statistical* sampling algorithm. The non-statistical sampling algorithm we propose considers information from previous accesses to determine which accesses can be ignored while generating the MRC and which accesses should not be ignored. Ignoring a set of accesses while generating the MRC reduces the processing overhead required to generate the MRC. However, the memory overhead for a non-statistical sampling algorithm is larger than the memory overhead for a statistical sampling algorithm, as additional information about the state of the cache is required for the algorithm to work.

At a high level, the non-statistical sampling algorithm we developed goes through the following phases when processing a workload:

1. Each accessed key is processed and an MRC is generated using one of the traditional MRC generation algorithms. This phase continues until the MRC has stabilized and is minimally changing.

2. Each accessed key is processed, not to update the MRC, but to identify changes in the workload's access pattern relative to the previous phase. The basic idea is that it takes less effort to identify a change in the access pattern than it is to continue updating the MRC. This phase continues until such changes to the access pattern have been identified.

3. Once a change to the access pattern has been identified, the algorithm returns to Phase 1.

This cyclic process is repeated indefinitely until the workload has been fully processed.

Based on our experimental work, we show that our non-statistical sampling algorithm generates very accurate MRCs in most cases while offering an improvement in throughput. By setting the parameters of our algorithm, we can choose to decrease the accuracy of the MRC which increases the throughput of the algorithm. This leads to having up to a 5.5% loss in throughput for MRC generation when compared to a currently existing statistical sampling algorithm while achieving improved and acceptable MRC accuracy. We define the acceptable MRC accuracy to be less than 2%, as mentioned by CounterStacks and AET [12, 28]. In some cases where workloads show large changes to the MRC and the access pattern during runtime, the performance is similar to or worse than Olken because the non-statistical sampling algorithm cannot sample accesses when the MRC is changing most of the time. The accuracy of the MRC remains acceptable because the accesses are processed similar to Olken, which is an exact algorithm. The memory overhead is reduced when dealing with workloads that show minimal change to the MRC and increased when dealing with workloads that show a large change to the MRC.

The main problem with the non-statistical sampling algorithm we propose here is that the algorithm cannot sample out accesses when the workload access pattern is constantly changing. When this happens, the algorithm effectively behaves as Olken does, processing and updating the MRC after each individual access. One untested idea to fix this problem is to apply a statistical sampling algorithm such as SHARDS during phase 1 of our algorithm, which would have performance better than Olken but worse than SHARDS because the algorithm still has to check if the workload access pattern has changed. As future work we further need to explore different ways of determining when the access pattern of the workload has changed so as to improve the performance of the algorithm.

This dissertation is structured as follows. First, in Chapter 2, we present the background and literature review. In particular, we describe in detail the currently existing MRC generation algorithms. In Chapter 3, we describe the proposed non-statistical sampling algorithm used for generating approximate MRCs. In Chapter 4, we report and analyze the experimental results. Finally, in Chapter 5, we review the proposed algorithm and draw conclusions.

Chapter 2

Background & Related Work

In-memory, DRAM-based caching systems are used to cache data stored in backend storage systems that use SSDs or HDDs. If the data requested by an application exists in the cache then the read access time is reduced because it is served out of DRAM. The load on the backend storage system is also reduced because a portion of the read accesses are served from the cache, which potentially reduces the load and thus access times of the backend storage system. As such, in-memory caching servers are prevalent in today's distributed systems. Two popular open-source in-memory caching systems are Redis [21] and Memcached [8] which store data as key-value pairs. The key is used to determine what value within the database is to be obtained. Each key is only associated with one value and is represented by an arbitrary string that may define a URL, a filename, a hash, or any other data. Because the key is used to access a value from the database, a value cannot be used to filter or control any results that are returned from an access to the database.

The basic key-value store operations are: get, put, and delete. The get(key) access returns the value associated with the key from the cache if the key is present, and returns a miss otherwise. A put(key, value) access adds the key-value pair into the cache if the key has never been seen before, or updates the value if the key already exists in the cache. A delete(key) access deletes the key-value pair if the key exists in the cache. The simplicity in the basic operations allows the key-value model to be fast, simple to use, and scalable.

In-memory caches typically operate as look-aside caches. When a get(key) results in a miss, the application fetches the data from the backend storage and issues a put(key, value) to the cache. When data is modified or newly written, the application updates the data in the backend storage and simultaneously invalidates the data in the cache that corresponds to the key which had its data modified. While Memcached only supports basic key-value data, Redis supports a larger variety of data types, including but not limited to String, Hash, and List, which allow for different structuring of data depending on the application.

The amount of DRAM allocated to an in-memory cache is crucial to the effectiveness of the cache. Having a cache be too small can result in a high miss rate, while having a cache be too large can result in superfluous memory utilization if the same miss rate can be achieved with a smaller DRAM footprint. A high miss rate can significantly increase the data access response time. For example, a 10% increase in miss rate can increase the response time by 300x or more [13]. On the other hand, a large unused DRAM footprint is costly due to the steadily increasing price of memory [11].

Miss rate curves (MRCs) are the only known tool to help decide on an effective cache size for a given trace of a workload. MRCs plot the percentage of cache misses as a function of the cache size for a given workload. By using the information from an MRC, one can determine if increasing the size of the cache is necessary to obtain a target miss rate or if decreasing the size of the cache is an available option without increasing the miss rate significantly. Before Mattson et al.'s creation of a single pass MRC generation algorithm, generating an MRC was achieved by running a separate cache simulation for the specific cache size of interest. Running a single cache simulation provides a single point of data on a complete MRC for the specified cache size. Therefore, multiple simulations, each requiring high processing and memory overheads, had to be performed to obtain an MRC that can be used to select an effective cache size for the given workload.

2.1 Fixed Sized Data Block Algorithms

2.1.1 Mattson

The issue of running separate cache simulations for generating an MRC was resolved in 1970 by Mattson et al. when he introduced an algorithm that generates an MRC using only a single pass over a trace of the target workload's data accesses [15]. By maintaining a stack of the accessed keys ordered by access recency, the algorithm generates a histogram of stack distances from which the MRC can be derived. The histogram is separated into evenly distributed buckets. This results in less overhead when updating the histogram at the cost of a minimal loss of accuracy. The stack distance for a given access key is the number of distinct keys that were accessed since the last time the current key was accessed. Since the stack is ordered by access recency, the operation performed is to search down the stack until the current key is found. For a given stack distance value, the stack distance histogram is updated by adding one into every bucket that has a smaller or equal value to the stack distance value. If the current key is not found in the stack, then the key is being encountered for the first time and is represented with a stack distance of infinity in the histogram (essentially adding to every bucket in the histogram). Using the stack distance, one can determine whether the currently accessed key would result in a cache miss or not for any given cache size. For a given fixed data value size d with a stack distance of s, the current access would result in a cache miss if the resultant $d \times s$ is larger than the configured cache size c, and a cache hit otherwise.

While Mattson is the first to achieve a single pass algorithm for generating MRCs, the algorithm exhibits processing overheads of O(NM) for processing N accesses and M unique accessed keys as a result of having to search through the stack of uniquely accessed keys for every accessed key. Mattson's algorithm also requires a memory overhead of O(M) memory to hold the stack of uniquely accessed keys, resulting in less available memory for caching additional data. Mattson also assumes that a least recently used (LRU) eviction policy is applied. For all further discussion we assume that the eviction policy is LRU.

2.1.2 Olken

One optimization to Mattson's algorithm that improves the processing overhead is featured in Frank Olken's algorithm, aptly named Olken [18]. Olken's algorithm makes use of a generalized AVL tree to store the uniquely accessed keys instead of a stack, while still maintaining order by access recency. The same procedure is done where the currently accessed key is searched for in the tree. If found, the key is removed from the tree and reinserted as the rightmost child to reposition it as the most recently accessed key. Using AVL trees guarantee a maximum tree height and searching operation of O(log M), and thus delete and reinsert have overheads of O(log M) for a total processing overhead of O(log M). This leads to a processing overhead of O(N log M) for the Olken algorithm which outperforms the processing overhead of O(NM) exhibited by Mattson's algorithm. Note that the memory overhead remains the same as the AVL tree will hold all uniquely accessed keys. The procedure for calculating the stack distance is the same as in Mattson's, where the stack distance for a given access key is the number of distinct keys that were accessed since the last time the current key was accessed.

2.1.3 SHARDS

Instead of changing the type of data structure as Olken did for Mattson's algorithm, SHARDS aims to improve the efficiency of Mattson's algorithm through spatial sampling. SHARDS [27] uses Mattson's or Olken's algorithm to generate an MRC, but instead of doing a single pass over the entire trace of the target workload's data accesses, it only performs a single pass over a randomly sampled subset of the data accesses, effectively reducing the size of the list of ordered accessed keys used. An accessed key r is sampled if the condition $hash(r)\ mod\ 100 < K$ is met for a specified sampling rate $K \epsilon$ [0, 1]; resulting in the algorithm sampling *(K*100)%* of the entire trace. While the processing and memory overheads are still O(NM) and O(M) respectively, SHARDS requires considerably less overhead to generate an MRC depending on the value of K. One additional property of SHARDS is that it ensures all accesses to a sampled key will be included throughout the trace. While this is an attractive property, it also has a downside that if a frequently accessed key is not sampled, it will never be sampled throughout the trace despite its high frequency.

Further evaluation of the SHARDS algorithm shows that the accuracy of approximating MRC's using spatial sampling is dependent on the values of N and M. For small values of N and M, it is better to use a large value of K to improve the accuracy of the MRC, while for large values of N and M, it is better to use a small value of K to limit the memory required. By adaptively lowering the sampling rate from a high value at the start of the single pass over a trace, one can maintain a fixed bound on the total number of sampled keys. This is accomplished by setting a maximum desired size for the set S of sampled keys, where each key in the set also has information about its associated *threshold value (hash(r) mod 100).* When a key is first sampled and inserted into S, if the bound on the size of S is exceeded, then the key in the set S with the largest *threshold value* is removed. In effect, the removal of the key whose *threshold value* represented the highest sampling rate value lowers the overall sampling rate in the set S. This allows the sampling rate to be lowered from a high value at the start of the single pass over a trace. Applying an adaptive sampling rate with a fixed number of sampled keys allows for the generation of an MRC with processing and memory overheads of O(N) and O(1) respectively.

Another observation of SHARDS is that for the cases where SHARDS exhibits non-trivial error, the difference is mostly accounted for by a coarse "vertical shift" of the MRC, while the finer

	a	b	b	c	**b**	a	d	c	a	a
c_1	1	2	2	3	3	3	4	4	4	4
c_2		1	1	2	2	3	4	4	4	4
c_3			1	2	$\boxed{2}$	3	4	4	4	4
c_4				1	$\boxed{2}$	3	4	4	4	4
c_5					1	2	3	4	4	4
c_6						1	2	3	3	3
c_7							1	2	3	3
c_8								1	2	2
c_9									1	1
c_{10}										1

Figure 2.1: Example of the operation of the CounterStacks algorithm.

features of the MRC are modeled accurately. This led to the development of $SHARDS_{\text{adj}}$, which is an adjustment of the generated MRC at minimal cost that improves accuracy significantly. This is done by shifting the parts of the MRC that are separated by a vertical shift, minimizing the portions of the MRC which account for the largest error.

2.1.4 CounterStacks

CounterStacks uses a completely different approach. Instead of maintaining an exact list of the accessed keys, CounterStacks periodically instantiates counters which record the number of distinct keys that have been accessed since the inception of the counter. In the most basic form of the algorithm [28], these counters are instantiated on every accessed key, and all previous counters are incremented by one if and only if they have not previously seen the newly accessed key. The stack distance used to generate the MRC is then derived by traversing the counters in historical order from the oldest and searching for the first occurrence of two consecutive counters, c_i, c_{i+1}, where the counter c_i does not increment while counter c_{i+1} does increment. This means that the counter c_i has encountered the key that is currently being processed while the counter c_{i+1} has not. This results in a non-infinite stack distance because the currently processed key has been processed at or before the i^{th} access, but has not been processed at the $(i+1)^{th}$ access.

Figure 2.1 shows an example of the CounterStacks algorithm. In the third instance that key b is accessed, counter c_3 does not increment while counter c_4 increments by one. This identifies that key b has been accessed at the third historical access or earlier, but not at the fourth access. To determine the stack distance for the key b at position 5 whose previous reference was at position 3, we take the value $C_{3,5}$ to obtain a stack distance of 2.

The basic form of CounterStacks has a processing and memory overhead of O(N^2M) and O(N) respectively. The algorithm is very inefficient due to two problems. The first is that each counter must record all distinct keys it has encountered and the second is that the search operation performed on each counter must be done on every accessed key.

To solve the first problem, CounterStacks uses HyperLogLog (HLL) probabilistic counting to approximate the number of unique keys for each counter [9]. An HLL counts the number of distinct keys using a fixed amount of memory and eliminates the need to maintain a list of previous accesses. HLLs achieve this by estimating the cardinality of a set of uniformly distributed random numbers by calculating the maximum number of leading zeroes in the binary representation of each number. If the maximum number of leading zeroes is n, then the HLL will estimate the number of distinct elements in the set as 2^n. In practice, the referenced keys are hashed to obtain a more random distribution before estimating cardinality, and references are partitioned into subsets and combined using the harmonic mean to improve the accuracy of the counters [10]. To solve the second problem, CounterStacks downsamples the counters so that they are only instantiated and updated every d accesses, reducing the total number of counters from N to N/d.

These two optimizations reduce the processing overhead to O(N^2) and memory overhead to O(Nlog(N)). CounterStacks' experiments pick the downsampling value to be d = 10^6, making the processing overhead of O($(\frac{N}{10^6})^2$) which is small in comparison to Mattson's processing overhead. In addition to downsampling, CounterStacks also prunes younger counters when their value is identical to the next older counter. In this scenario, the younger counter provides no additional information compared to the older counter because the two counters will remain the same forever, allowing for the deletion of the younger counter. This further reduces CounterStacks' processing overhead to O($(\frac{N}{d})^2$log(M)).

2.1.5 AET

AET uses yet another approach. Instead of maintaining a histogram of stack distances, AET maintains a histogram of reuse time, where reuse time of an accessed key is defined as the time between the current access and the previous access to the same key [12]. AET also maintains a LRU stack for incoming accesses. The data corresponding to a key k will move down one position in the LRU stack if and only if the stack position of the currently accessed key *k'* is greater than k's stack position, or if the currently accessed key results in a cache miss. As a result, the velocity of k or the speed at which k moves down the LRU stack at time x is equal to *P(t)* which is the probability of an access where its reuse time t is greater than x. Computing *P(t)* for each t in linear time is trivial given a reuse time histogram and will produce T_c (as the reuse time histogram tells us when a block is evicted), where T_c is the eviction time for a given cache size c. The eviction time is the time between a key's last access and its eviction from the cache. Knowing T_c, the miss rate curve for a given cache size c is $MRC(c) = P(T_c)$. Because generating the MRC is dependent on the reuse time histogram and its probability distribution, generating a sampled reuse time histogram with similar distribution to the exact reuse time histogram can improve processing and memory overheads for maintaining the reuse time histogram while approximating an accurate MRC [12].

SHARDS and CounterStacks are approximate algorithms because they apply sampling to reduce the processing and memory overheads of their respective algorithm. AET, by sampling the reuse time histogram, can also achieve similar reduction in processing and memory overheads, also making it an approximate algorithm. These algorithms use statistical sampling, where the frequency of sampling is dependent on a set of sampling parameters. The issue with statistical sampling is that no prior information of the trace is used to determine whether an incoming key being accessed for the first time should be sampled or not. Problems such as the one mentioned in SHARDS, where a frequently accessed key may not be sampled and will never be sampled, can arise, possibly introducing errors in the generated MRC [27].

2.1.6 Zhong and Chang

Zhong and Chang introduced a new sampling technique based on dividing the trace into repeated sequences of two interleaving intervals: a sampling interval and a hibernating interval [33]. This sampling technique is closely related to our work that we describe and evaluate in subsequent sections.

In the naive scenario, all accessed keys in the sampling interval are sampled while all accessed keys in the hibernating interval are ignored. Any keys being sampled will generate a stack distance that contributes to the stack distance histogram, which generates the MRC for a given trace using Mattson's algorithm. This scenario has no guarantee of accuracy, since an accessed key can have a large stack distance error if it was accessed within a hibernating interval but was skipped for sampling.

To solve the accuracy problem, Zhong and Chang introduced *biased sampling*, where accessed keys in the hibernating interval do not update the MRC, but are still checked to see the last time the key was accessed. If the key was accessed in the same hibernating interval, it is ignored, and the processing of the trace continues. If the key was accessed for the first time or accessed before the current hibernating interval, the key is sampled. However, the issue with *biased sampling* is that keys with a short stack distance in the hibernating interval will not be sampled, resulting in the sampling algorithm favoring keys with large stack distances.

Zhong and Chang thus introduced *history-preserved representative sampling* as a fix. *History-preserved representative sampling* works like *biased sampling*, except the stack distance is only measured if the first access of the sampled key falls within a sampling interval. This change makes the probability of a key being sampled the same regardless of their stack distance values. *History-preserved representative sampling* exhibits over 99% accuracy in their experiments and is thus effective. However, the additional sampling required throughout the hibernating intervals adds additional overhead that the hibernating intervals did not have in the naive scenario.

The algorithm we present in Chapter 3 is similar to Zhong and Chang's naive scenario where there is an interleaving of sampling and hibernating intervals. The difference between our algorithm and the naive scenario is that the trace is divided into the two interleaving intervals based on the change in the workload access pattern instead of being based by a fixed number of accesses. By having hibernating intervals occur only when the workload access pattern is minimally changing, we are able to obtain accuracy results similar to the *history-preserved representative sampling* but without the extra overhead of having to sample additional accesses throughout the hibernating intervals.

2.1.7 Low Cost Working Set Size Tracking

Low cost working set size tracking [32] generates LRU-based MRCs to model the relationship between physical memory allocation and page misses. To reduce the overhead of maintaining the MRCs, a technique called intermittent memory tracking (IMT) is used to lower the overhead without a significant loss of accuracy. The idea is similar to the idea by Zhong and Chang: the execution of a program is divided into phases. Within a phase, the working set size remains nearly constant. This allows for the memory tracking to be temporarily disabled to avoid tracking cost within a stable phase. When a phase change is predicted to occur based on a change in the working set size, the memory tracking is re-enabled until there are no more predicted phase transitions.

Our algorithm is very similar to the intermittent memory tracking technique, which also uses the

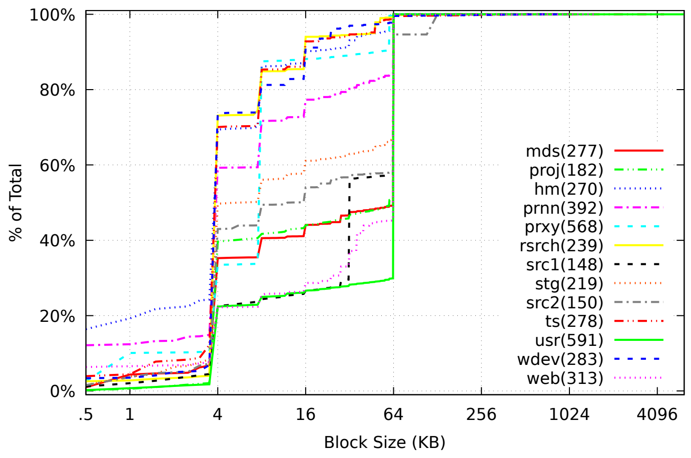

Figure 2.2: Cumulative distribution function for the different traces from the MSR collection. (Between brackets is the number of block sizes used by the trace

working set size criteria as a method of identifying a phase change. However, low cost working set size tracking focuses on page-level LRU-based MRCs and also use hardware events such as data TLB misses, L1 cache misses, and L2 cache misses to infer their decision on whether a phase change has occurred. Our algorithm focuses on in-memory caching systems and uses different criteria available to us instead of hardware events for making our phase change decisions.

2.2 Variable Sized Data Block Algorithms

The algorithms described above assume that the size of each referenced data block is the same. Given that DRAM-based caches can allocate memory proportional to the size of the data being cached and many real-world workloads include data blocks of different sizes, our research group has determined that the size of the data for each accessed key matters when generating an MRC. Figure 2.2 shows that MSR traces [16] all have different data block size distributions; on average each trace uses 300 different block sizes.

One way to generate an MRC for variable sized data is to assume all referenced data blocks have the same size (i.e. 4KB, 16KB, or the average of all referenced data blocks in the trace) while another is to divide each referenced data block into fixed sized data blocks that add up to the original size (i.e. dividing a 11KB data block into three references of 4KB data blocks each). While prior work use either method of generating an MRC given variable sized data, both strategies are highly inaccurate in comparison to the MRC generated using exact block sizes. Figure 2.3 shows

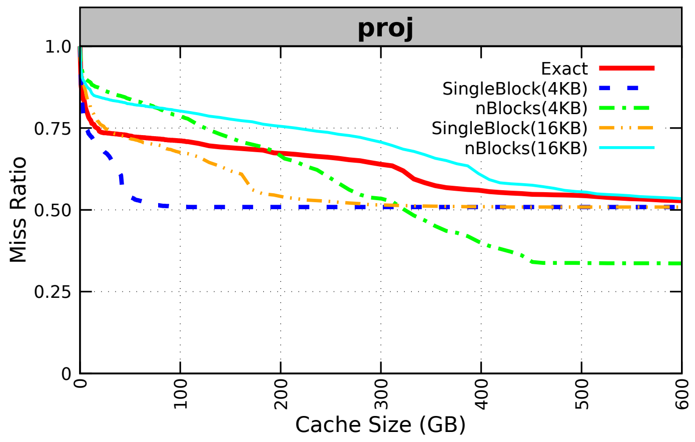

Figure 2.3: Miss rate curves for different block size treatment for the ”proj” workload from MSR traces.

the inaccuracy for generating an MRC for the `proj` workload from the MSR traces using the two explained methods compared to the exact MRC.

Our research group adapted some of the above MRC generation algorithms to take variable sized data blocks into consideration. Mattson’s algorithm required two changes to the base algorithm. The first change is to include the size of the data block for each accessed key in the stack element being recorded, increasing the memory required by a constant factor. The second change is to the definition of the stack distance for an accessed key. Instead of the stack distance defining the number of distinct keys accessed since the current key was last accessed, the stack distance is now defined as the size of memory required to store data from all distinct keys accessed since the current key was last accessed. The stack distance histogram is generated similar to the base algorithm, except without multiplying the number of distinct keys by a fixed data block size. As a result, the processing overhead for Mattson’s algorithm remains the same.

Because SHARDS uses the same stack distance histogram methodology as Mattson’s algorithm to generate an MRC, the changes required for Mattson’s algorithm are also the same changes required for SHARDS. The existence of variable sized data blocks does not change the sampling methodology used in SHARDS, and as a result the processing overhead will remain the same while the memory overhead will be increased by a constant factor.

To accommodate variable sized data blocks in CounterStacks, the counter must now maintain a list of values, where each value in the list is a counter for the number of distinct accessed keys for

	A(4k)	x(2k)	B(4k)	C(4k)	y(2k)	A(4k)	x(2k)	y(2k)	**B(4k)**	B(4k)
c_1^{4k}	1	1	2	3	3	3	3	3	3	3
c_1^{2k}	0	1	1	1	2	2	2	2	2	2
c_2^{4k}		0	1	2	2	3	3	3	3	3
c_2^{2k}		1	1	1	2	2	2	2	2	2
c_3^{4k}			1	2	2	3	3	3	$\boxed{\mathbf{3}}$	3
c_3^{2k}			0	0	1	1	2	2	**2**	2
c_4^{4k}				1	1	2	2	2	$\boxed{3}$	3
c_4^{2k}				0	1	1	2	2	2	2
c_5^{4k}					0	1	1	1	2	2
c_5^{2k}					1	1	2	2	2	2
c_6^{4k}						1	1	1	2	2
c_6^{2k}						0	1	2	2	2
c_7^{4k}							0	0	1	1
c_7^{2k}							1	2	2	2
c_8^{4k}								0	1	1
c_8^{2k}								1	1	1
c_9^{4k}									1	1
c_9^{2k}									0	0
c_{10}^{4k}										1
c_{10}^{2k}										0

Figure 2.4: Example of the modified CounterStacks algorithm to support variable block sizes.

a specific data block size since the counter was instantiated. The stack distance used to generate the MRC is, similar to the base algorithm, derived by traversing the counters in historical order from the oldest and searching for two consecutive counters where a specific block size of the first counter c_i does not increment while the second counter c_{i+1} does increment. The stack distance is the sum of the old counter values multiplied by each counter's block size. As with the fixed size data block CounterStacks algorithm, using HLL probabilistic counting and downsampling the counters will improve the overheads of the variable size data block CounterStacks algorithm.

Figure 2.4 shows an example of CounterStacks processing a trace of variable sized data blocks. Each accessed key generates a counter c_i which is a list containing two counters; a counter for incoming 4KB data blocks and a counter for incoming 2KB data blocks. Looking at the second access of the key B with a data size of 4KB, the third instantiated counter c_3^{4k} remains at a count of three while the fourth instantiated counter c_4^{4k} increases from two to three, meaning the currently accessed key has been seen previously before. The stack distance is then calculated as the sum of the counters multiplied by the counter's block size, which is (4KB multiplied by 3) + (2KB multiplied by 2) = 16KB.

All our experimental work described in this dissertation takes into account variable block sizes,

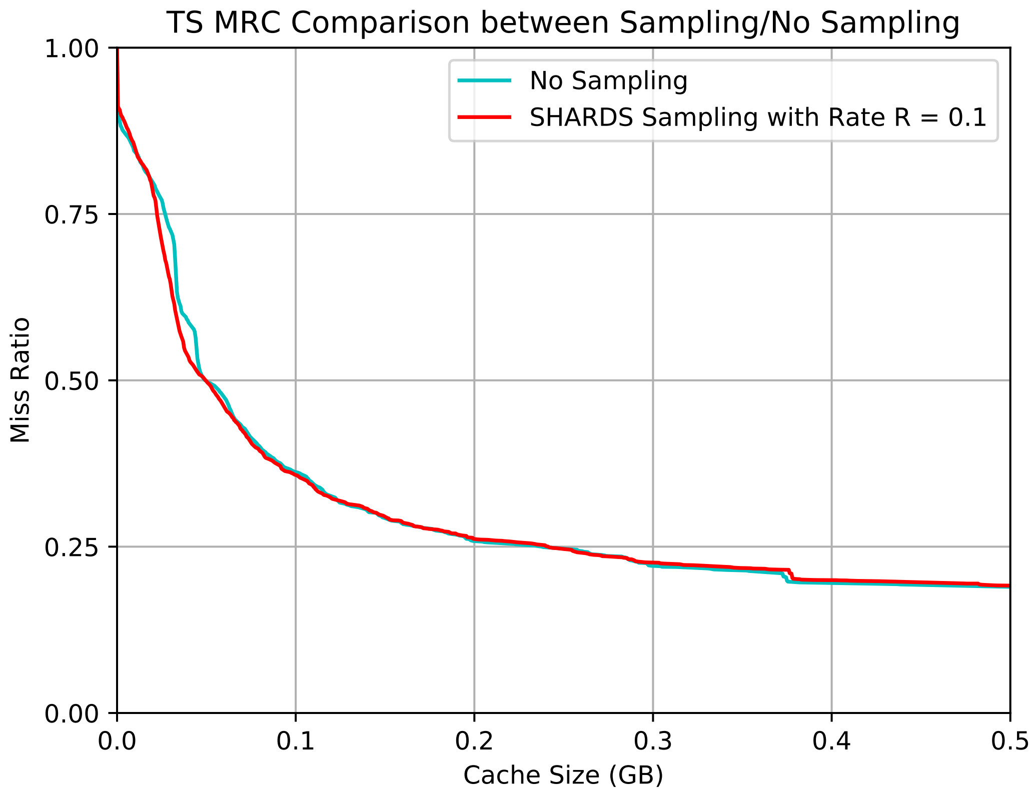

Figure 2.5: Comparison of MRCs generated with sampling and without sampling for the *ts* workload.

and the MRCs generated are compared against exact MRCs for variable block sizes.

2.3 Measuring Errors in the Generated MRC

SHARDS, CounterStacks, AET, and Zhong and Chang use some form of sampling to lower processing overheads for generating MRCs. This introduces errors into the MRCs that are generated. In the literature, the standard method of calculating the error for an MRC is to use the mean absolute error, or MAE for short. The MAE is calculated as $\frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$, where x_i is the miss ratio of data point i on the accurate MRC, y_i is the miss ratio of data point i on the approximate MRC, and n is the number of data points along the MRC.

The MAE describes the average error across the entire MRC, including errors in the MRC at small cache sizes where the miss ratio tends to be higher. Most users would find the high miss ratios unacceptable and are hence interested in the area of the MRC where the miss ratios are acceptable. This leads to another method of calculating the MAE where only data points along the MRC are considered that are below a given miss ratio requirement. Figure 2.5 shows the comparison between an exact MRC and an approximate MRC generated by using SHARDS for the `ts` workload. Calculating the MAE over 1000 evenly distributed data points along the MRC results in a MAE of 0.625% for the *ts* workload. However, if the user has a miss ratio requirement of 50% or less, then the MAE for the *ts* workload is 0.356%, which is only 57% of the full MRC error.

A tertiary method of calculating the MAE is to only use data points up to a maximum cache size of interest.[1] In public cloud environments, memory for a cache such as Microsoft's *Azure Cache for Redis* [20], is often sold at pricing tiers based on the amount of memory. More memory increases the pricing tier and the cost per month for the memory. A user can limit themselves to a specific pricing tier and use the MAE up to the maximum cache size for that tier to consider the error and whether it is acceptable for their application. Using the same *ts* workload example, the MAE starting at a cache size of 0 and going up to a maximum cache size of 0.3GB would be 0.831%, which is larger than the full MRC error.

In other cases, one may be interested in the maximum error rather than an average, allowing a user to determine whether the worst case miss rate error is within the error bounds acceptable to the user. Analyzing the *ts* workload again, the maximum error would be 9.22%, which is a result of using a cache size of 31MB.

Despite the availability of the alternative error metrics described above, we will exclusively use MAE in the following sections because (i) it is the error metric used in publications of related work, and (ii) the alternatives require parameters that depend on the objectives of the users of the caches. In our calculations of MAE, we include all points up to the cache size after which no further improvements of the miss rate occur.

[1] It is possible to mislead the reader if the approximate MRC at large cache sizes is quite accurate, by calculating the MAE for cache sizes up to unreasonable sizes. The MAE can then be improved simply by calculating the MRC for larger and larger cache sizes.

Chapter 3

MRC Generation with Non-Statistical Sampling

Statistical sampling can be defined as drawing a set of observations randomly from a population distribution [22]. Statistical sampling originates from audit sampling, in which the financial records of a company are examined and verified for correctness [2][7]. Because the number of records that must be verified are very large, audits are sampled to not waste resources. Statistical sampling utilizes a statistical method such as random sampling to select which records are verified. Inversely, non-statistical sampling is based on the auditor's judgement and on a set of criteria. Only sampling records where the value of items exceed $100,000 is an example of a non-statistical audit sampling method.

In this chapter, we will be presenting a new non-statistical sampling algorithm for MRC generation. This algorithm uses a set of criteria to determine whether to sample an access or not when processing the workload. Section 3.1 will discuss the issues in recent MRC generation algorithms that use statistical sampling. Section 3.2 goes over our new sampling algorithm at a high level. Section 3.3 will describe each phase of the algorithm in detail and includes a discussion on various criteria that will be used in the algorithm.

3.1 Statistical Sampling Issues

A key issue in SHARDS and AET is that they use key-based statistical sampling techniques. These algorithms will not sample a large set of keys due to the method of sampling. Key-based statistical sampling samples across the keys, which include the property that once a key is sampled, it will always be sampled in future accesses. Keys that are not sampled can affect the calculated stack distance for future accesses, leading to inaccurate MRCs.

A contrived simple example can explain the issue with statistical sampling. Consider the following sequence of accesses:

A, b, c, d, **F**, d, c, b, **A**, b, d, c, **F**, c, b, d **A**

If we assume the size of the cache is 20KB and each key references a 5KB block of data, then two keys periodically evict each other out of the cache. At every occurrence other than the first access, key A evicts key F and key F evicts key A. Applying Matton's algorithm on the given order of key accesses result in a miss rate of 47% at the assumed cache size of 20KB. If a statistical sampling algorithm naively sampled only the keys A, b, c and d and not key F, the generated MRC would be inaccurate at the point where the cache size is equivalent to the assumed cache size of 20KB. This is because all key-value pairs belonging to the 4 sampled keys will fit in the cache of size 20KB, and no evictions would be recorded. Applying Mattson's algorithm on the *sampled* key accesses results in a miss rate of 27% for the assumed cache size of 20KB.

Although the above is a contrived example, these type of scenarios occur in practice. Consider Figure 3.1, which shows a significant error when statistical sampling is used. The MRC is generated for the Microsoft Research `web` workload [16] in two ways: with no sampling and with SHARDS at a sampling rate of R = 0.001. The SHARDS paper mentions that for a fixed-rate sampling of R = 0.001, the approximate MRCs have a median MAE of less than 2%, and that the MAE is bounded by a maximum error of 5%. However, when applying the SHARDS algorithm for the *web* workload example, the MAE is 7.7% and the maximum error is 37%. Figure 3.1 shows unaligned cliffs where the miss ratio drops substantially within a small increase in cache size. This problem may lead to provisioning an incorrect amount of DRAM for a target miss rate, such as configuring 62GB of DRAM to achieve an expected 45% miss rate, when configuring 50GB of DRAM would reach the

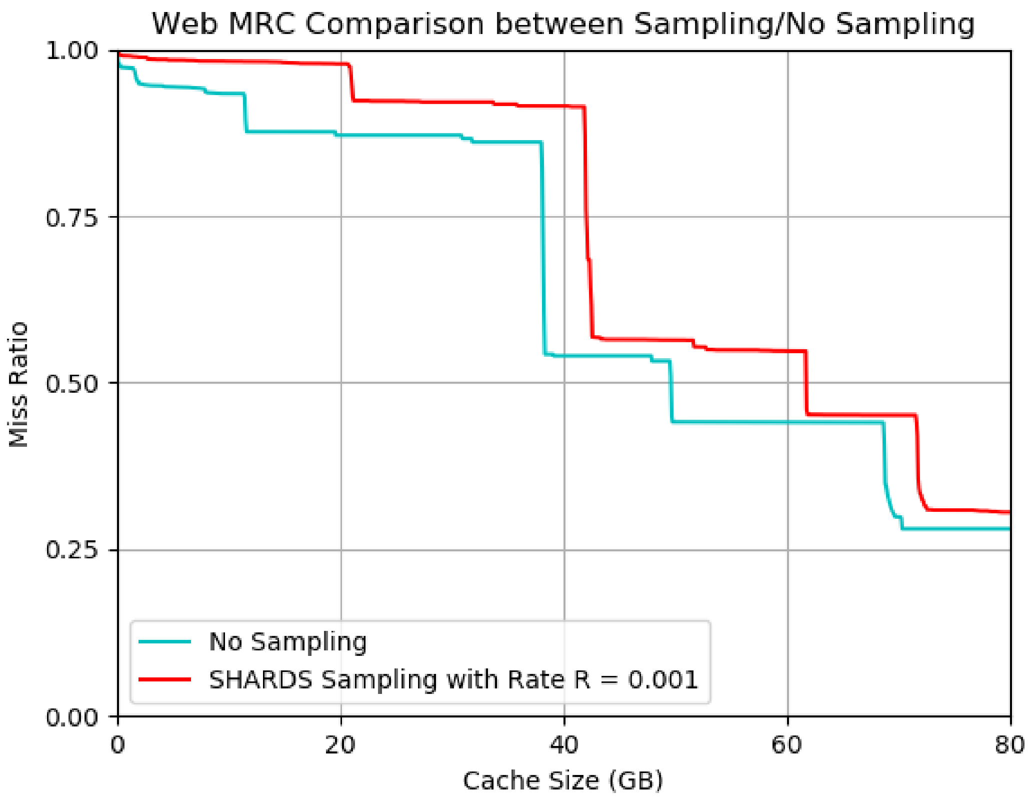

Figure 3.1: Comparison of MRCs generated with sampling and without sampling for the *web* workload.

targeted 45% miss rate according to the exact MRC.

Table 3.1 shows that when applying SHARDS with a sampling rate of R = 0.001 on the *web* workload, the 5 most frequently accessed keys are not sampled. In fact, the most frequently accessed key that is sampled, web-0-2100141568, is only the 381st most frequently accessed key in the workload. While the total number of accesses of the 381 most frequently accessed keys is less than 1% of the total number of accesses for the *web* workload, the non-sampled keys that are accessed with high frequency cause additional errors in the approximate MRC. By increasing the fixed-rate sampling from R = 0.001 to R = 0.01, the web-0-18475052544 key becomes the most frequently accessed key with 205 accesses. MAE comparisons for the two given sampling rates also shows that the higher the sampling rate, the lower the MAE of the approximate MRC. For example, increasing the sampling rate from R = 0.001 to R = 0.01 causes the MAE to drop from 7.7% to 6%.

No Sampling		SHARDS with R = 0.001	
Key	Frequency	Key	Frequency
web-0-2099764736	1499	web-0-2100141568	65
web-1-1511288320	1413	web-0-19735432704	24
web-1-1426824704	1234	web-0-2065636864	14
web-1-1426820608	1232	web-0-17001541120	11
web-1-1426939392	1231	web-0-18572426752	10

Table 3.1: Comparison of Most Frequently Accessed Key for Web Workload

3.2 Algorithm Overview

The remainder of this chapter introduces a non-statistical sampling algorithm that samples key accesses using information from prior accesses. While still a sampling method that will result in an approximate MRC, we will show that the approximations made by our non-statistical algorithm will, on average, be more accurate than statistical sampling, but at the cost of extra compute and memory overheads.

Our non-statistical sampling algorithm is called PR-MRC or Pattern Recognition MRC that, at a high level, iterates over two phases and works as follows:

1. **Convergence Phase:** Process each access to update the MRC and at the same time capture the access pattern of the workload; this is done until the MRC converges.

2. **Access Pattern Monitoring Phase:** Process each access, not to update the MRC, but to capture the access pattern; this is used to identify a change in the access pattern.

3. When a change in the access pattern has been identified, return to **Convergence Phase**.

Our hypothesis is that while the access pattern remains the same, the MRC does not need to be updated because the MRC will not change significantly from its converged state. This reduces compute overhead because the cost of updating an MRC is far more expensive than identifying changes to the access pattern, which will be shown in Section 3.3.

There are different candidate methods to identify changes in the access pattern. They include but are not limited to:

- Reuse time histograms.

- Comparison of data points from an MRC of the same workload generated earlier in time.

- Frequency of commonly accessed keys.

For example, when identifying changes in the access pattern, one might consider monitoring the frequency of commonly accessed keys. If a key is frequently accessed over a period of time but is then accessed infrequently in the next batch of accesses, one might conclude that the access pattern has changed.

3.3 Low Level Algorithm Discussion

In this section we will present the step by step procedure for both the *Convergence Phase* and the *Access Pattern Monitoring Phase*. Section 3.3.1 describes the *Convergence Phase*, including defining what is convergence and how to determine when we have reached convergence. Section 3.3.2 describes the *Access Pattern Monitoring Phase*, discussing how we monitor changes in the access patterns and various criteria that are used to help us monitor changes.

3.3.1 Identifying MRC Convergence

In the *Convergence Phase*, each access is processed using a traditional, exact MRC generation algorithm. In our implementation, we use Olken because it is the most efficient of the exact MRC generation algorithms. Based on the hypothesis described in Chapter 3.2, we want to be able to identify when an MRC has converged. To do so, we partition the accesses into fixed size intervals, and at the end of each interval, we take a snapshot of the MRC. We then compare the MRCs taken from two adjacent intervals to determine whether the MRC has not changed significantly and hence whether convergence has been achieved.

The size of the interval, I_{CP}, is a key parameter. If the interval is too small, then the MRCs of two adjacent intervals being compared will more likely be very similar, leading our method to incorrectly conclude that the MRC has converged when it has not. If the interval is too large, then it may take too long to conclude that the MRC has converged, making PR-MRC less efficient. By default, we selected the size of the interval to be one million accesses. We present a sensitivity analysis on this parameter in Chapter 4.

We propose two methods of comparing MRCs from two adjacent intervals: *error convergence* and *slope convergence*. For the *error convergence* method, the MRC is considered to have converged if the average absolute difference between corresponding points on the two curves is less than a threshold

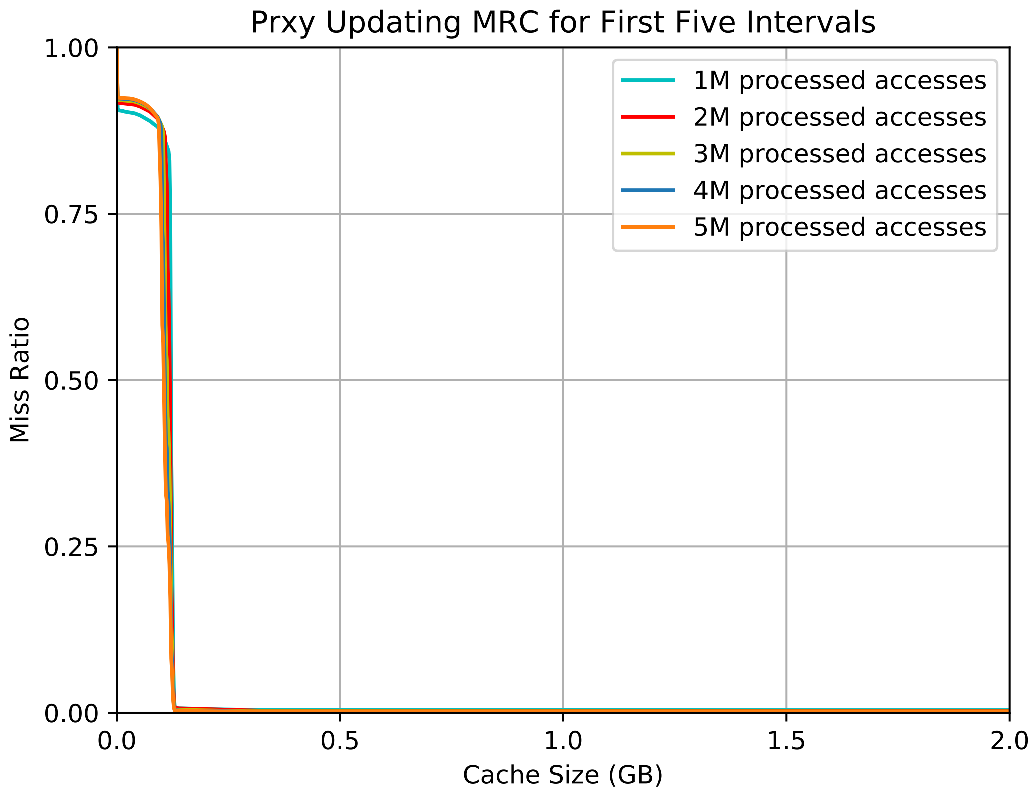

Figure 3.2: Captured MRCs for *prxy* workload used in the error convergence method.

value v_{th}. Thus, the two MRCs are compared by taking the difference in miss ratio at multiple points along the curve. The threshold value is a parameter that is used to determine whether the MRCs being compared are sufficiently similar.

Figure 3.2 is an example that shows five captured MRCs generated for the first five intervals of the *prxy* workload using the default interval size of one million accesses. Algorithm 1 provides the pseudocode for comparing MRC values along 1000 equidistant points along the cache size axis for two captured MRCs. For each ordered data point on the MRC, the absolute difference in miss ratio between the two data points are added up. The average is calculated and compared to the threshold value v_{th} to check for convergence.

Table 3.2 shows the average absolute miss ratio difference between each MRC and the MRC at the previous interval for the same *prxy* workload; that is, it shows the average error expressed in miss ratio. By setting the threshold value $v_{\text{th}} = 1\%$, we can conclude that the MRC for the *prxy*

```
Algorithm 1: Error Convergence Method
Data: Current MRC miss ratio cMRC[1000]; Previous MRC miss ratio pMRC[1000]
Result: Boolean conv = True: True indicates convergence, False indicates non-convergence
Input: Threshold Value $v_{\text{th}}$
double avg = 0;
for $i \leftarrow 0$ to 999 do
    avg += | $(cMRC[i] - pMRC[i])$ |;
if $\frac{avg}{1000} > v_{th}$ then
    conv = False;
```

# Processed Accesses	Average Absolute Miss Ratio Difference
2,000,000	0.29%
3,000,000	0.30%
4,000,000	0.20%
5,000,000	0.16%

Table 3.2: Calculated average absolute miss ratio difference for *prxy* workload used to determine convergence.

workload will have converged after processing two million accesses.

Selecting the threshold value parameter has a significant effect on deciding whether or not the MRC has converged. Increasing the threshold value relaxes the requirements necessary to achieve convergence, which will reduce the time taken to achieve convergence. However, the workloads we tested indicate that this will result in a higher error in the MRCs updated in subsequent intervals and in the final MRC generated by PR-MRC. This occurs because an MRC that is being compared to the MRC at the previous interval can converge with a larger average absolute difference in miss ratio.

If we decrease the threshold value, the requirements to achieve convergence become more strict. This means it will take longer to achieve convergence, but the error in the MRCs updated in subsequent intervals will decrease. This approach is the direct opposite of the increased threshold value scenario, as an MRC that is being compared to the MRC at the previous interval can converge only with a smaller average absolute difference in miss ratio.

An issue in identifying a converged state using the *error convergence* method is that some workloads will exhibit no convergence at all. An example is the *web* workload shown in Figure 3.3, where the MRC exhibits cliff-like behaviors with steep drops in the miss ratio at specific cache sizes. Using Algoirthm 1 with the default interval size, the average absolute difference in miss ratio between the

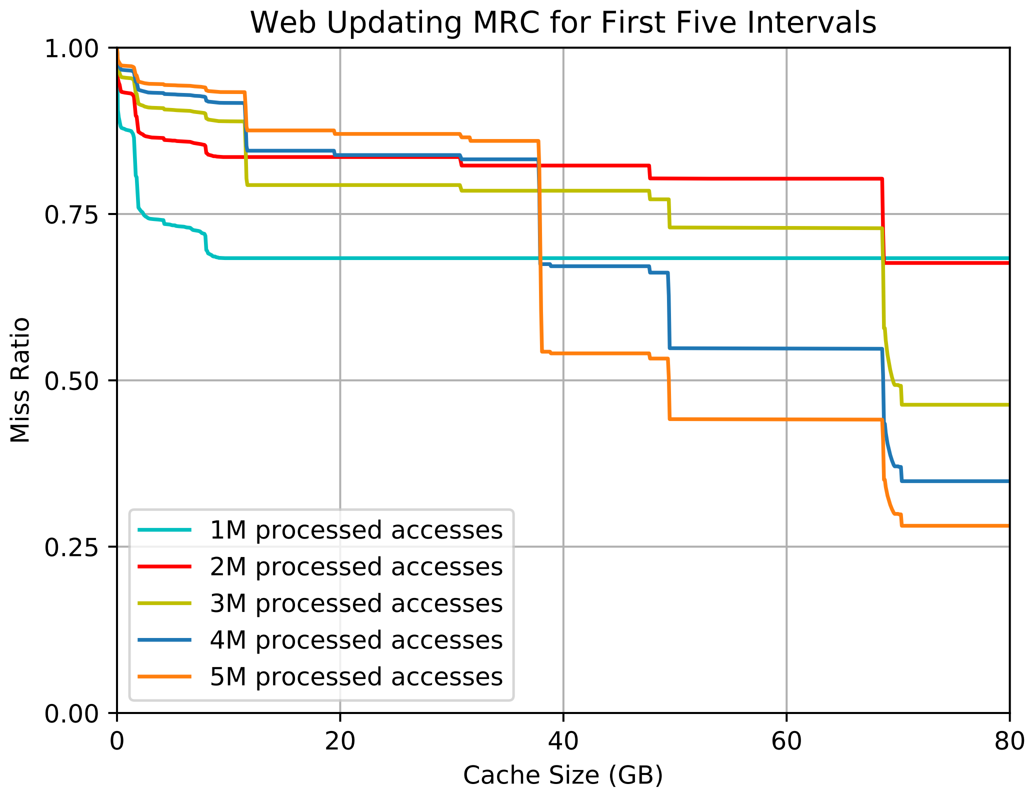

Figure 3.3: Captured MRCs for *web* workload used in the slope convergence method.

MRC generated after 4 million accesses and 5 million accesses of the workload is 6.64%. One might consider setting the threshold value to a high value such as 10%, causing the convergence check to be less strict. However, this solution becomes less effective as the MRC may not converge regardless of the threshold value.

This prompts the second method of identifying convergence called the *slope convergence* method. The *slope convergence* method compares the slopes of two MRC curves generated in sequence. The basic idea behind this method is based on an observation in our experimental work, namely that if an MRC has cliffs with steep drops, the position of the cliffs will converge early in the workload. The slope convergence also uses a threshold value v_{th} as its only parameter. The threshold value is used to determine whether the slopes of the MRCs being compared are sufficiently similar.

For workloads that exhibit cliffs in their MRC, the operational objective is to select the size of the memory based on the position of the cliffs; that is, to select a cache size that corresponds to a

Algorithm 2: Slope Convergence Method

```
Data: Current MRC cMRC[1000]; Previous MRC pMRC[1000]
Current Approximate Slope cSlope[999], Previous Approximate Slope pSlope[999]
Result: Boolean conv = True: True indicates convergence, False indicates non-convergence
Input: Threshold Value $v_{\text{th}}$
for $i \leftarrow 1$ to 999 do
    $cSlope[i] \leftarrow |\, (cMRC[i] - cMRC[i-1])\, |$;
    $pSlope[i] \leftarrow |\, (pMRC[i] - pMRC[i-1])\, |$;
    // Slope Difference = cSlope[i] - pSlope[i]
    if $|\, (cSlope[i] - pSlope[i])\, | > v_{th}$ then
        $conv = False$;
```

point immediately to the right of the cliff. As long as the position of the cliffs are accurate, then the overall accuracy of the MRC is less critical for the user. In fact, as shown in Figure 3.3, the miss ratio of the MRC segments between cliffs may be substantially off using the *slope convergence* method.

The *slope convergence* method calculates the slope across 1000 equidistant points along the MRC. The slope is calculated using the equation $m = \frac{y_i - y_{\text{i-1}}}{x}$, where y_i and $y_{\text{i-1}}$ is the miss ratio for two consecutive points along the MRC while x is the difference in cache size between the two points along the x-axis of the MRC. Since the 1000 points along the MRC are all equidistant, calculating the division in the slope equation is not necessary. Therefore, we simplify the calculation to compare only the numerator of the slope equation $y_i - y_{\text{i-1}}$ across the MRC. The slope difference is the absolute difference between the two calculated slopes with the same index value from the current MRC and the previous MRC in sequence. If the maximum slope difference is below the threshold value v_{th}, then the MRC has achieved a converged state.

Algorithm 2 provides the pseudocode for confirming whether the MRC has achieved a converged state using the *slope convergence* method. The maximum slope difference between two consecutive intervals for the *web* workload is shown in Table 3.3. Similar as with the *error convergence* method, increasing the threshold value relaxes the requirement to achieve convergence, while decreasing the threshold value makes the requirement to achieve convergence more strict.

While both convergence methods can be applied to any workload, the *error convergence* method will generate an approximated MRC with less error to the true MRC as opposed to using the *slope convergence* method. The *slope convergence* method is effective only on MRCs that contain cliffs because of the observation that the cache size corresponding to the cliffs converge early in the

% Processed Accesses	Maximum Slope Difference
2,000,000	7.88%
3,000,000	5.08%
4,000,000	10.68%
5,000,000	1.56%

Table 3.3: Calculated maximum slope difference for *web* workload used to determine convergence

workload. By determining the position of the cliffs, the client can select the cache size corresponding to the cliff plus extra cache size to cover any inaccuracies in the approximate MRC which could place the client on the wrong side of the cliff. Since the position of the cliffs along the MRC converge early, we can determine their position by searching for very steep slopes early in the convergence phase. Therefore, the *error convergence* method is used at the start of the convergence phase. If a set of cliffs have been observed, then the algorithm can switch to the *slope convergence* method for the remainder of the convergence phase and any further convergence phases for the processed workload.

3.3.2 Identifying Changes in the Workload Access Pattern

In this section we describe the step by step procedure for the *Access Pattern Monitoring Phase* of PR-MRC that follows the *Convergence Phase*. During the monitoring phase, the MRC is not updated, and hence the LRU stack is also not updated. The objective of the *Access Pattern Monitoring Phase* is to determine when to switch back to the *Convergence Phase* so as to continue updating the MRC while doing so with lower overhead than that required to update the MRC.

We partition the accesses to be monitored into fixed size intervals. In each interval, we capture the characteristics of the accesses that might identify an access pattern. We then compare the captured characteristics of the latest interval and the one immediately previous to it to determine whether the access pattern has changed.

We consider three criteria that might allow us to identify the characteristics of the access pattern. These three criteria are:

1. **Working Set Size:** The number of unique bytes accessed in the interval. More specifically, the sum of the sizes of the value part of each accessed key-value pair (in bytes), where the key is being accessed for the first time in the interval.

2. **Approximate Miss Rate:** The approximate miss rate in an interval on a small simulated cache that does not evict keys; once the cache is full, no new objects are added.

3. **Key Popularity:** The set of the n-most accessed keys in the interval. To identify workload changes, we consider the size of the intersection of the two sets from the two most recent adjacent intervals.

We also considered other criteria, and combinations thereof, but we found them to be ineffective. Some of the other criteria we considered were accesses per second, read/write ratio, and the size distribution of the value component of accessed key-value pairs. We did not find these criteria to be helpful in determining whether the access pattern has changed or not, as the results were either too different from one interval to another, or showed very little change even if the access pattern changed. As such, we have dropped these criteria from consideration and will only move forward in discussion with the three criteria listed above.

The size of the monitoring interval, I_{MP}, in terms of the number of accesses, is a key configuration parameter. If the interval is too small, there may not be enough accesses to identify an access pattern. If the interval is too large, PR-MRC may identify a combination of access patterns in one interval rather than capturing multiple access patterns that may be the same given a smaller interval, making PR-MRC switch back to the *Convergence Phase* too often. By default, we selected the size of the interval to be 1 million accesses in our experimental results, so it is the same size as that used in the *Convergence Phase* $I_{\text{CP}} = I_{\text{MP}}$. We present a sensitivity analysis of this parameter in Chapter 4.

PR-MRC also monitors the access pattern in each interval of the *Convergence Phase*. This is done so it can compare the characteristics obtained in the first interval of the *Access Pattern Monitoring Phase* with the characteristics obtained from the previous interval which is part of the *Convergence Phase*. Figure 3.4 shows this. Each colored box represents a different workload access pattern. The three green adjacent boxes identify a similar access pattern. A red box is shown on the next interval, signifying that the access pattern has changed from the green box. Because the adjacent green and red boxes have different access patterns, PR-MRC returns to the *Convergence Phase*. When PR-MRC moves back to the Monitoring Phase at a later interval, a new access pattern is found, shown with purple boxes. This access pattern remains the same until it reaches the interval represented by the yellow box, representing another access pattern. Once again, PR-MRC returns to the *Convergence Phase* as a result of the two different access patterns.

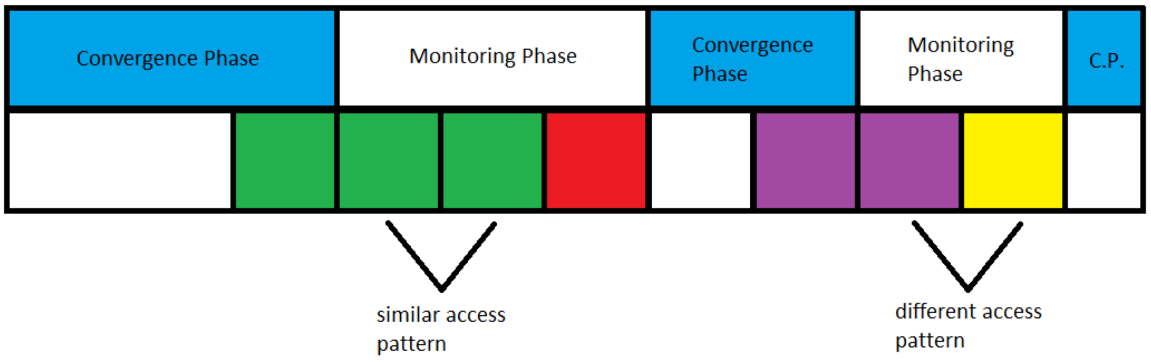

Figure 3.4: Methodology of the *Access Pattern Monitoring Phase*.
Each colored box represents an access pattern.

Access Interval	Working Set Size Difference
407M - 408M	3.43%
408M - 409M	3.86%
409M - 410M	3.19%
410M - 411M	2.72%
411M - 412M	3.55%
412M - 413M	2.91%
413M - 414M	2.79%

Table 3.4: Working Set Size Difference across multiple Twitter Cluster25 workload intervals.

The first criteria we describe in detail is the *working set size* criteria. For this criteria, we sum up the number of bytes in the value part of accessed key-value pairs for keys being accessed for the first time in the interval.[1] Our motivation for using the *working set size* as a criteria is that it has been established that when programs go from one phase to another phase, their working set size changes [4, 31, 32]. We assume that when a phase change occurs, the memory access pattern changes.

To compare the working set size from two adjacent intervals, we take the percentage difference between the two values. Given an interval A with a working set size WSS_A, and the adjacent interval B with a working set size WSS_B, the percentage difference between the two values is calculated as $\frac{|\mathrm{WSS}_A - \mathrm{WSS}_B|}{\mathrm{WSS}_B} \cdot 100$ for $\mathrm{WSS}_A < \mathrm{WSS}_B$, and $\frac{|\mathrm{WSS}_A - \mathrm{WSS}_B|}{\mathrm{WSS}_A} \cdot 100$ for $\mathrm{WSS}_A > \mathrm{WSS}_B$. If the percentage difference between the working set sizes is less than or equal to a threshold value v_{th} parameter, then the working set sizes are deemed to be similar, indicating that the access pattern has not changed.

[1] We experimented with taking the sum of the number of bytes in the value part of accessed key-value pairs for keys being accessed for the first time in the entire workload, but the results were worse than only considering per interval unique accesses.

Table 3.4 show the percentage difference between working set sizes between the accessed interval and the previous interval for seven adjacent intervals starting from a random interval in the Twitter `Cluster25` workload. All of the percentage difference between working set sizes from one interval to the next show similar results and, given a low threshold value, would confirm that the access pattern is not changing across the seven intervals. By default, we selected the threshold value $v_{\text{th}} = 5\%$ for the majority of our experiments.

The second criteria we consider to determine whether the access pattern has changed or not is the *approximate miss rate*. We simulate a small cache of fixed size that does not evict any keys. Thus, once the cache is full, every incoming access is treated as a cache miss if it is not in the cache. Our motivation for using the miss rate of a simulated small-sized cache is the assumption that different access patterns have different miss rates. Our motivation for using the *approximate miss rate* as a criteria on a non-evicting cache, as opposed to simulating a traditional LRU cache with evictions, is that the latter has higher processing and memory overheads and our experiments indicate that both approaches are equally effective. Simulating the full cache behavior requires the maintenance of the recency stack, whereas the approximate miss rate can be obtained without the recency stack.

Two parameters are used to determine whether the access pattern has changed using the *approximate miss rate*. The first parameter is the size of the simulated cache, *cSize*. The *cSize* cannot be too small or too large, as the miss rate we are generating is a data point on the MRC for the interval. If the *cSize* is too small, then the miss rate will be very high, and the miss rate will likely be similar even if the access patterns are quite different. If the *cSize* is too large, it is possible that the cache holds the entire working set size and the miss rates are not representative of the access pattern.

By setting the *cSize* to an initial value of 10MB, we can dynamically increase or decrease the size of the simulated cache at the end of an interval if the results show that the *cSize* is too small or too large. However, doing this requires us to capture the criteria for two new intervals when there is a change in the *cSize* so the approximate miss rate comparison is fair.

The second parameter is the threshold value v_{th} which, similar to the *working set size* criteria, is used to compare the approximate miss rate percentage between two adjacent intervals.

Table 3.5 show the approximate miss rates for seven adjacent intervals starting from a random interval in Twitter Cluster7. The *cSize* dynamically increased to 160MB prior to the starting interval in the table. The result show that the approximate miss rates are all similar from one interval to the

Access Interval	Approximate Miss Rate
210M - 211M	1.47%
211M - 212M	1.09%
212M - 213M	1.03%
213M - 214M	0.88%
214M - 215M	0.75%
215M - 216M	0.64%
216M - 217M	0.54%

Table 3.5: Approximate miss rate where *cSize* = 160MB across multiple Twitter Cluster7 workload intervals.

next. Thus we can determine that the access pattern has not changed throughout these accesses.

The third criteria, *key popularity*, generates a set of the n-most accessed keys for each interval. To identify a change in the access pattern, we take the intersection of the two sets from the two most recent intervals. If the size of the intersection is below a threshold, then we deem that the access pattern has changed.

Two parameters are used for the *key popularity* criteria. The first parameter is the size of the set of the number of keys we are considering, n. We generate the set by maintaining a hashtable that keeps track of the count of each incoming access, which adds memory overhead. If n is too large, then the set may include many non-popular keys. If n is too small, the set will not include many popular keys. In our experiments, we use a $n = 10$ because we have noticed that in most cases, the ten most popular keys account for over 70% of the accesses in the interval. The second parameter is the threshold value, v_{th}, used to determine if the size of the intersection is low enough to declare that the access pattern has changed.

Table 3.6 shows an example of the *Key Popularity* criteria for two adjacent intervals starting from a random interval in Twitter Cluster25. Given $n = 10$, the size of the intersection between the two sets is ten, as all keys from the first interval appear in the second interval. In fact, the next twenty intervals after the 1.819B - 1.820B interval all show the same 10-most frequently accessed keys, showing that the access pattern is very stable and not changing for this portion of the workload.

Accesses 1.818B - 1.819B		Accesses 1.819B - 1.820B	
Key	Frequency	Key	Frequency
2405856321318320000	318766	2405856321318320000	326057
14598119693547000000	161329	14598119693547000000	163218
15044292555629500000	129498	15044292555629500000	132922
5849555380574390000	113900	5849555380574390000	111044
8893207267468060000	51763	8893207267468060000	51592
15911355532589300000	45097	15911355532589300000	43329
14845985722235800000	19357	14845985722235800000	19023
14859065261456500000	11070	14859065261456500000	10456
6511662763440140000	11070	6511662763440140000	10456
16508893134111400000	5159	16508893134111400000	4878

Table 3.6: Comparison of the set of the 10-most frequently accessed keys between two adjacent intervals for the Twitter Cluster25 workload.

Chapter 4

Evaluation

We ran PR-MRC on a wide variety of workloads to evaluate the effectiveness of PR-MRC. In this chapter, we:

1. compare different configurations to select a single configuration of PR-MRC that is compared against SHARDS;
2. show how accurate PR-MRC is when compared against Olken, an exact MRC generation algorithm, and SHARDS, another approximate MRC generation algorithm described in Chapter 2.1;
3. compare the throughput of PR-MRC versus Olken and SHARDS; and
4. compare the memory overhead of PR-MRC versus Olken and SHARDS

Section 4.1 describes the experimental setup used in our evaluation, while Sections 4.2, 4.3, 4.4, and 4.5 discuss each aforementioned evaluations in detail.

4.1 Experimental Setup

Workloads: For our evaluation, we used publicly available workloads from the three different sources listed in Table 4.1. The first source is the Microsoft Research Lab (MSR) trace set [16] containing 13 workloads totalling 434 million accesses. The second source is the U.S. Securities and Exchange Commission (SEC) trace set [6, 23] containing 58 workloads in their EDGAR log file data

Workloads	workloads	accesses
MSR Cambridge [16]	13	434M
Twitter [30]	54	247B
SEC EDGAR [6, 23]	58	25B

Table 4.1: Workloads used for evaluation.

set, totalling 25 billion accesses. The third source is the trace set provided by Twitter [30] containing 54 workloads totalling 247 billion accesses.

In our experimentation, we test PR-MRC on 13 Twitter workloads. The selected Twitter workloads contain 2 workloads where SHARDS has been known to perform poorly, and 11 workloads recommended by the paper that introduced the Twitter workloads [30]. Our evaluation is done with orders of magnitude more accesses from publicly available workloads than previously published studies [12, 26, 27, 28]. We also use the MSR Cambridge workloads because they have been widely used in prior research [12, 26, 27, 28], even though the MRCs generated contain cliffs and are not representative of typical MRCs.

Variable Block Sizes: All of the workloads include variable-sized blocks. In the past, researchers have used two methods to accommodate variable-sized blocks. One method is to disregard the variable-sized blocks and treat all blocks as a fixed block size (eg. 4KB, 8KB, 16KB, or average of the variable block sizes) [27]. The other method used is to translate each variable-sized block into as many fixed-size blocks (eg. 4KB, 8KB, 16KB, etc.) as needed to include the data [5, 28]. However, our research team has concluded that both those approaches lead to highly inaccurate MRCs for some workloads, indicating that using the exact block sizes is important when generating MRCs.

In all our experiments, we take all variable block sizes into account, but we round the block size up to the next power of two, in part because it is the most commonly used method for in-memory caches [8, 14]. This reduces the number of distinct block sizes.

Operation Treatment: The workloads we used include a number of different access operations (e.g. GET/PUT/SET/etc.). Some of the operations only occur because of the specific cache size used when the workload was collected. Since an MRC covers all cache sizes into account, we only use the GET and READ requests from the workloads, as was done in previous studies [12, 28]. A key is added to the recency stack whenever an access in the trace is a miss or is seen for the first time. For the Twitter workloads, we extracted the block size from the previous SET/ADD/APPEND requests for the same key as that of the GET request. To convert MSR workloads to key-value store accesses we assume READs are GET requests, similar to the approach used in [19].

Error Metric: In our evaluation, we use the mean average error (MAE) to quantify errors, as this was widely used in prior studies [24, 25, 26, 27, 28]. The MAE is calculated as $\frac{\sum_{i=1}^{n} |y_i - x_i|}{n}$, where x_i is the miss ratio of data point i on the accurate MRC, y_i is the miss ratio of data point i on the approximate MRC, and n is the number of data points along the MRC. The value of n is equal to the number of buckets in the stack distance histogram. We compare points on the MRC up to the point where both MRCs being compared cease to change (i.e., reach steady state), up to a maximum cache size of 2TB.

Experimental setup and configurations: We ran our experiments on a server with an AMD ThreadRipper 3990x (64 cores) and 256 GB of DRAM (DDR4-3200MHz). Some studies added workloads to RAM disks to reduce IO overhead as in [12]; in this evaluation workloads were read from a Sabrent Rocket Q 8TB NVME SSD. However, I/O time is omitted from throughput calculations for all of the tested algorithms for fairness. The implementation of the MRC algorithms read the trace from disk into a buffer in memory, processing the operations in the buffer before the next batch is taken from disk. To exclude the I/O time, we stop the timer every time a read system call is issued, and resumed when the read system call is completed.

PR-MRC is written in C# 10 and uses Olken during the Convergence Phase described in Chapter 3.3.1. We also implemented SHARDS and Olken, which are used for comparison purposes, in C# 10. We verified the correctness of our SHARDS and Olken implementations by comparing the MRCs they generated with those previously published [18, 27].

For our sensitivity analysis, we run our experiments with the threshold value v_{th} for the Convergence Phase ranging from 0.01 to 0.05. The threshold value for (i) the *working set size* criteria ranges from $v_{\text{th}} = 2\%$ to 10%, (ii) *approximate miss rate* criteria ranges from $v_{\text{th}} = 3\%$ to 7%, and (iii) the *key popularity* criteria ranges from $v_{\text{th}} = 3$ to 7. In addition for the *key popularity* criteria, we vary the parameter that specifies how many keys to consider in the set from $n = 5$ to 10. We lastly vary the interval size $I_{\text{CP}} = I_{\text{MP}}$ from 100K to 10M.

Cluster 6	Cluster 7	Cluster 11	Cluster 17
Cluster 18	Cluster 19	Cluster 22	Cluster 24
Cluster 25	Cluster 29	Cluster 44	Cluster 45
Cluster 52			

Table 4.2: The thirteen specific Twitter workloads that we run our experiments on.

4.2 Configuration Analysis

PR-MRC can be configured to use one of three different criteria: *working set size*, *approximate miss rate*, and *key popularity*. Each criteria has its own set of parameters. Hence, we refer to a configuration as a criteria with a specific set of associated parameters. In this section, we evaluate the behavior of PR-MRC under different configurations. Specifically, we consider accuracy, computational overhead, and memory requirements. We show there is no universally "best" configuration; rather it involves a trade-off between accuracy and computational overhead.

Figure 4.1 shows the accuracy of 12 different PR-MRC configurations. Each column represents a specific configuration; and each column contains 13 points, one for each Twitter trace we considered. We use the Twitter traces for this analysis because the Twitter traces represent a wide variety of workloads. The specific Twitter traces that we selected are listed in Table 4.2. We selected the eleven traces that Twitter recommends when running experiments and also included two additional traces that have a large MAE when using SHARDS. The average of the thirteen MAE's is depicted with a red X in each column.

The figure shows that the *working set size* criteria results in the lowest average MAE across the Twitter traces, while the *approximate miss rate* and *key popularity* criteria result in notably higher MAE. All configurations result in an average MAE of less than 1% which is considered good. The worst case MAE for the *working set size* is 1.79%, 2.19% for the *approximate miss rate*, and 3.41% for the *key popularity*. As defined earlier, a MAE of less than 2% is considered an acceptable MAE [12, 28].

Figure 4.2 shows the throughput for the same PR-MRC configurations as above. The same Twitter traces used for the accuracy analysis are used for the throughput analysis. The figure shows that the *approximate miss rate* criteria results in the highest average throughput across the Twitter traces. The average throughput using the *approximate miss rate* criteria is 2.5 times better than the average throughput using the *working set size* criteria which has the lowest average throughput

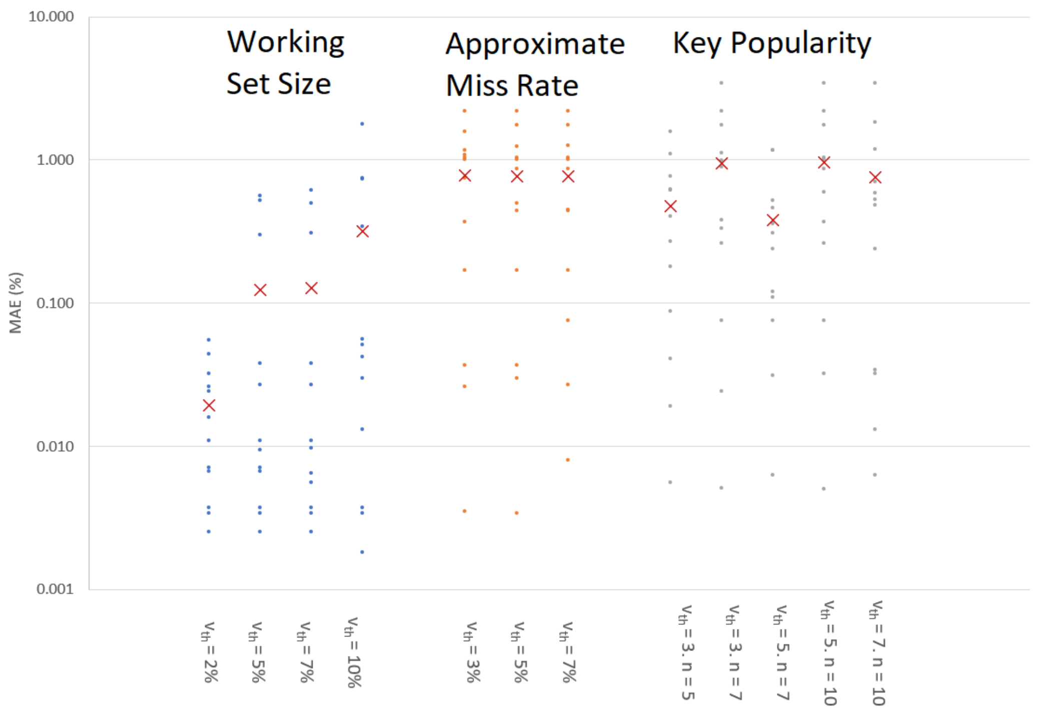

Figure 4.1: MAE of 12 distinct PR-MRC configurations tested on 13 Twitter traces, including the average MAE for each configuration shown as a red X. Note the logarithmic scale of the y-axis.

for all tested criteria.

Each criteria requires a certain amount of memory. For the *working set size* criteria, the memory requirement is O(M) for M uniquely accessed keys being seen for the first time in the interval. For the *approximate miss rate* criteria, the memory requirement is O(M) for M uniquely accessed keys being seen for the first time in the interval that fit in the simulated non-evicting small cache of fixed size. For the *key popularity* criteria, the memory requirement is O(M) for M uniquely accessed keys in the interval. The memory requirement of each criteria are similar enough. Thus we do not consider it further when trying to select the "best" configuration.

Figures 4.1 and 4.2 show that the MAE and the throughput are inversely correlated for the different configurations. The reason for this is that when PR-MRC stays in the *Access Pattern Monitoring Phase* for extended periods of time; (i) the MAE becomes worse because fewer accesses are used to generate the MRC, and (ii) the throughput increases for the same reason.

Given the fact that accuracy is inversely correlated to throughput, selecting the "best" configuration requires a judgement call. We argue that selecting the configuration that has the highest throughput while having an MAE less than 2% is a reasonable trade-off.

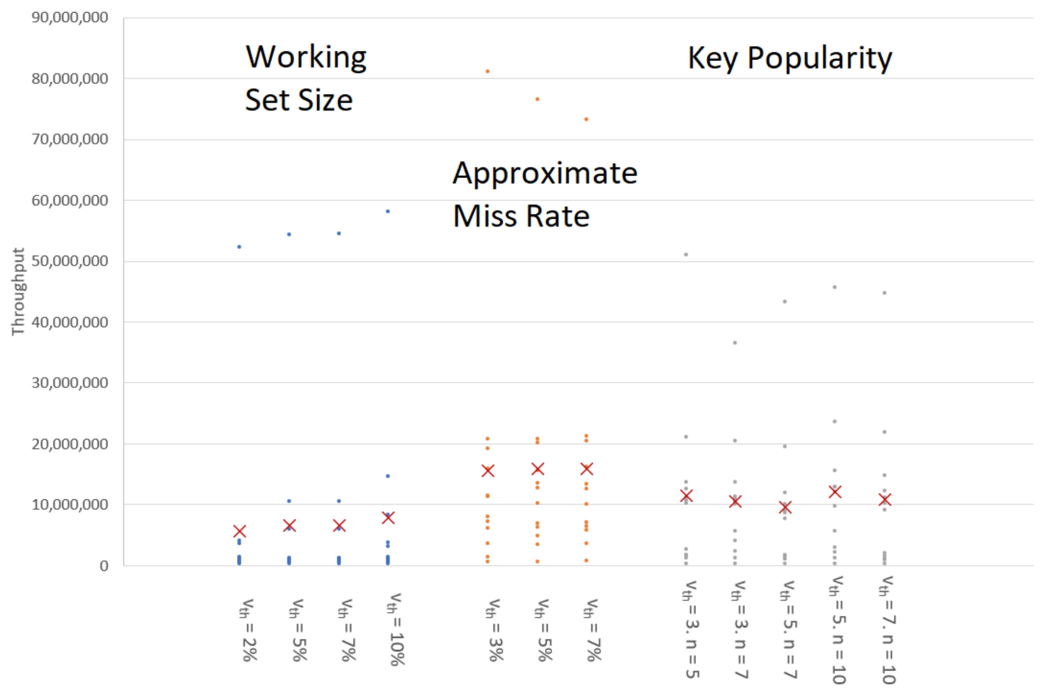

Figure 4.2: Throughput of 12 distinct PR-MRC configurations tested on 13 Twitter traces, including the average throughput for each configuration shown as a red X.

It is impractical to make this trade-off on a per-workload basis because it would require too much analysis. For this reason, our approach is to make the selection for the "best" configuration based on the average MAE and throughput across all workloads.

Figure 4.3 depicts a scatter plot showing the average MAE vs. the average throughput for each of the 12 configurations. Since the average MAE of all tested configurations are below 2%, we select the configuration that uses the *approximate miss rate* criteria with a $v_{\text{th}} = 5\%$ as the "best" configuration. This configuration will be used for the remainder of Chapter 4 when comparing against the MRC generated by SHARDS.

So far we have determined the "best" PR-MRC configuration where all tested configurations had the interval size parameter, I_{MP}, set to 1 million accesses. We now consider the effect of varying I_{MP} on the "best" PR-MRC configuration. Figure 4.4 depicts a scatter plot showing the average MAE vs the average throughput for the configuration that uses the *approximate miss rate* criteria with a $v_{\text{th}} = 5\%$. The figure shows that as I_{MP} increases, the throughput and MAE decrease. This is because when I_{MP} increases, more accesses are used to consider whether or not the access pattern has changed, increasing the possibility that the access pattern has changed. When I_{MP} decreases,

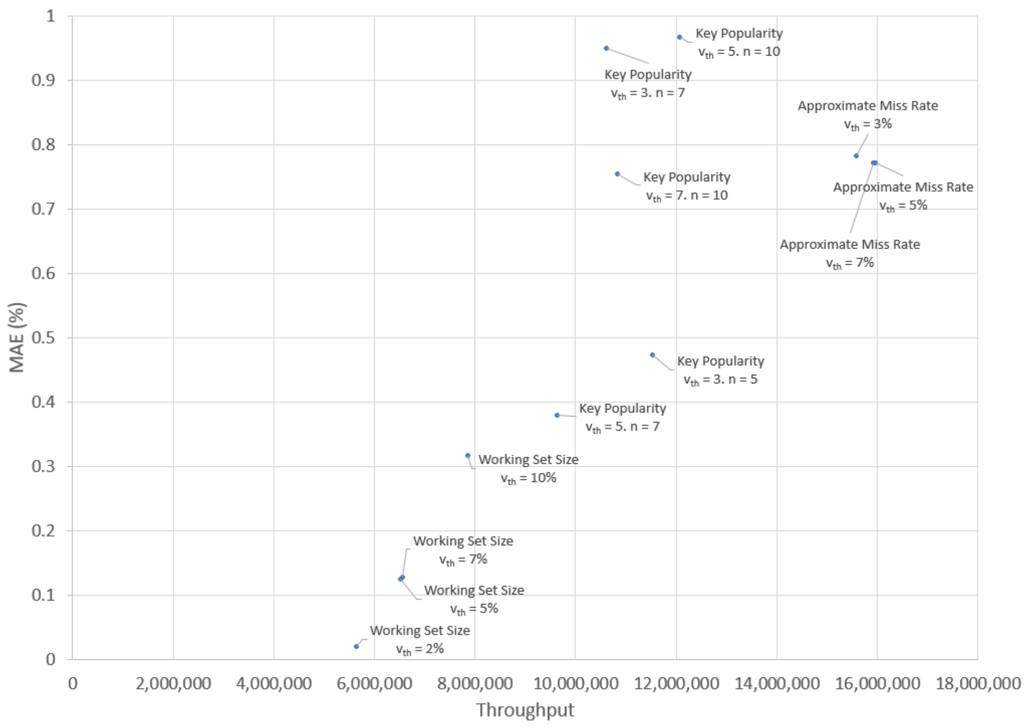

Figure 4.3: Average MAE vs Average Throughput of 12 distinct PR-MRC configurations tested on 13 Twitter traces.

the throughput and MAE increase because fewer accesses are used to consider whether or not the access pattern has changed, decreasing the possibility that enough of an access pattern has been identified to determine that the access pattern has changed. Since the configuration with an I_{MP} = 100K has a MAE less than 2%, our "best" configuration will use an I_{MP} = 100K for the remainder of Chapter 4.

4.3 Accuracy of PR-MRC

We compare the accuracy of the PR-MRC generated MRCs using the "best" configuration with SHARDS generated MRCs. An example where PR-MRC generated MRCs are more accurate than SHARDS is the `SEC 2017-QRT2` workload. This workload includes the second highest number of accesses across the SEC workloads at 2.15 billion accesses. Figure 4.5 plots (i) the exact MRC, (ii) the MRC generated by PR-MRC, and (iii) the MRC generated by SHARDS at a sampling rate of R = 0.1 and R = 0.01. The MAE for PR-MRC for the given workload is 0.008%, which is considered very good. The MAE of SHARDS using a sampling rate of R = 0.1 and R = 0.01 is 9.73% and

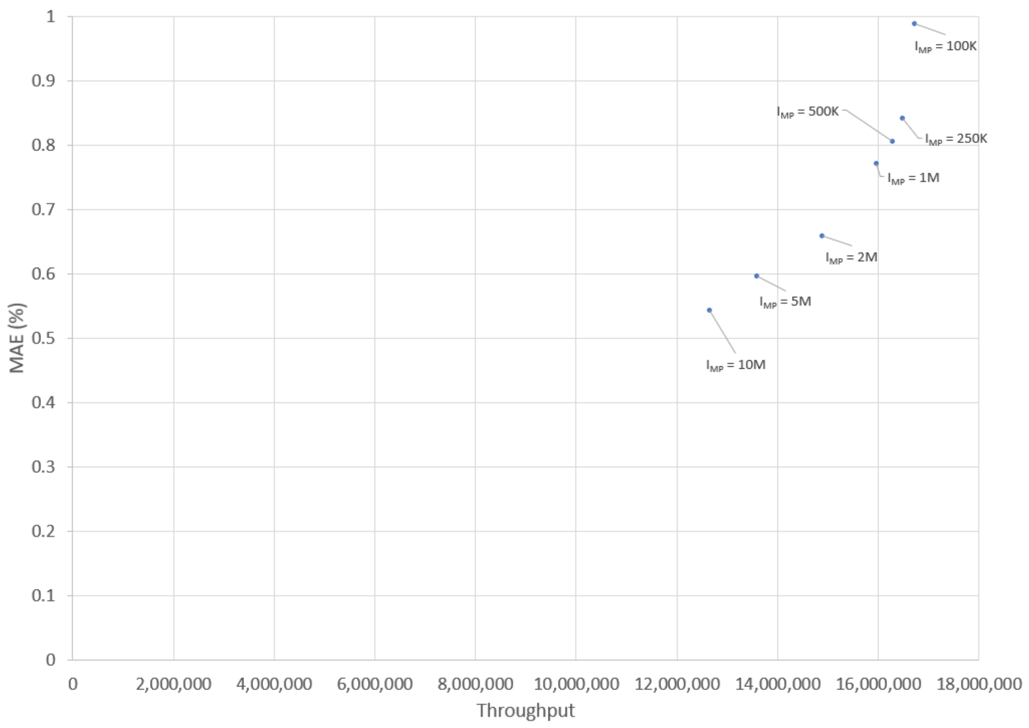

Figure 4.4: Average MAE vs Average Throughput using the "best" PR-MRC configuration and varying the interval size I_{MP} on 13 Twitter traces.

17.32%, respectively.

The reason PR-MRC outperforms SHARDS at both sampling rates is because the SHARDS algorithm may sample out very frequently accessed keys, which can skew the MRC that is being generated. Table 4.3 shows the ten most frequently accessed keys for the `SEC 2017-QRT2` workload when taking the exact MRC using Olken and when using SHARDS at a sampling rate of R = 0.1. None of the 10-most frequently accessed keys when running Olken appear when using SHARDS. The most frequently accessed key when running SHARDS is the 22nd most frequently accessed key when running Olken, and the 21 most frequently accessed keys constitute 56.7% of the total number of accesses in the workload. This many accesses that are sampled out cause the large error in the MRC generated by SHARDS.

An example where PR-MRC generates MRCs which are less accurate than SHARDS is the `SEC 2016-QRT3` workload. Figure 4.6 plots (i) the exact MRC, (ii) the MRC generated by PR-MRC, and (iii) the MRC generated by SHARDS at a sampling rate of R = 0.1 and R = 0.01. The *approximate miss rate* that is calculated stay within the threshold value of the "best" configuration. Given the

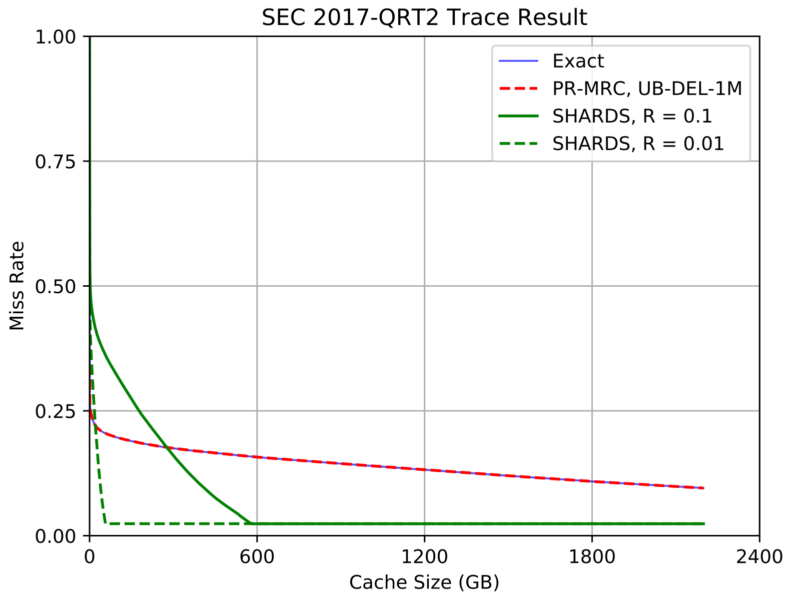

Figure 4.5: Comparison of MRCs generated by Olken, PR-MRC, SHARDS R = 0.1, and SHARDS R = 0.01 for `SEC 2017-QRT2` workload.

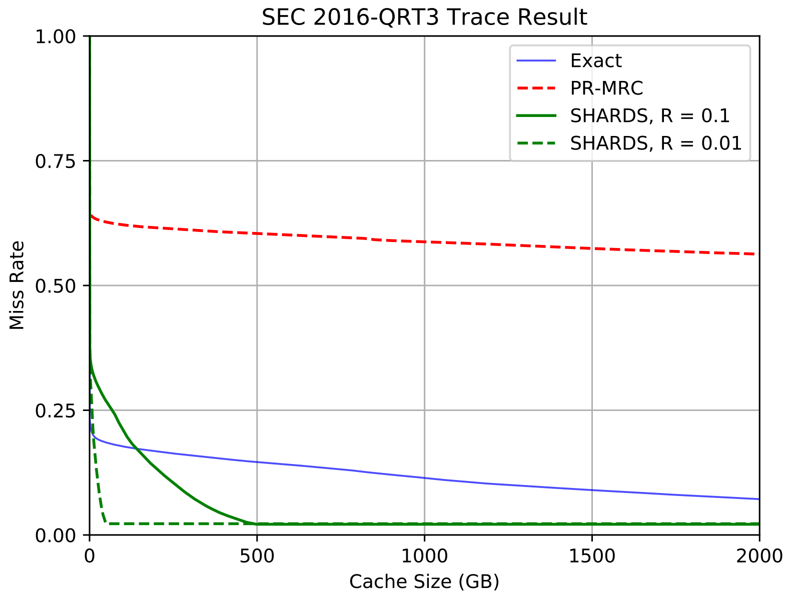

Figure 4.6: Comparison of MRCs generated by Olken, PR-MRC, SHARDS R = 0.1, and SHARDS R = 0.01 for `SEC 206-QRT3` workload.

Olken		SHARDS R = 0.1	
Key	Count	Key	Count
8761820731873650000	979865707	11875799237985600000	2565670
12759319744370900000	45679845	11785281799359000000	929522
7876360248967140000	44994505	17062651061906200000	810190
5004769155613920000	34091783	11132563875463200000	524238
1082831454404700000	15789838	942720673125417000	519696
15044020271529200000	9862149	14675856467279100000	306511
11088120511882600000	8727039	13057147722659100000	251266
7771191782240060000	8608610	5383526480106310000	249451
15331544608095300000	7983252	15830841463536800000	237634
9966625987577770000	7479302	6455280126504870000	213471

Table 4.3: Comparison of the set of the 10-most frequently accessed keys for Olken and SHARDS R = 0.1 for the `SEC 2017-QRT2` workload.

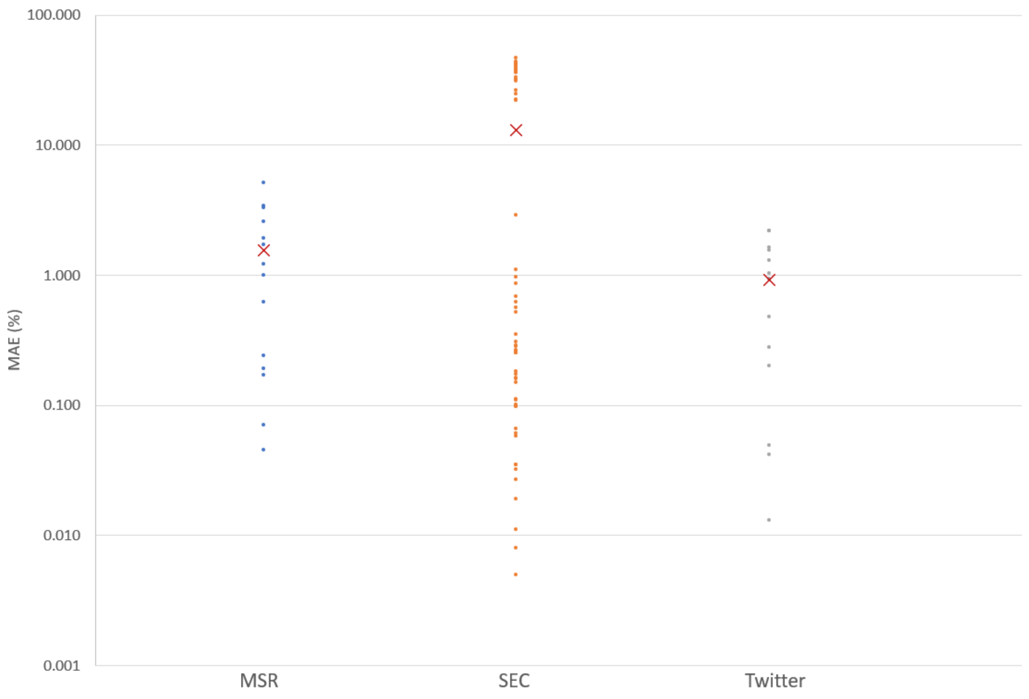

Figure 4.7: MAE of PR-MRC against Olken across all tested workloads, including the average MAE for each configuration shown as a red X. Note the logarithmic scale of the y-axis.

large error of 47.04%, PR-MRC incorrectly stayed in the *Access Pattern Monitoring Phase* as the access pattern was changing but was not identified by the chosen criteria used.

Figure 4.7 shows the MAE for all tested workloads across the MSR, SEC, and Twitter sets when using PR-MRC. PR-MRC results for the Twitter workloads show that the MAE across all

13 workloads result in an average of 0.99%. This is because most Twitter workloads exhibit a similar pattern when generating the MRC, where the MRC does not fluctuate after processing a small percentage of the overall workload. One example is the `Twitter Cluster18` workload, which contains more than 11.6 billion accesses. Figure 4.8 plots the exact MRC generated by Olken after 25% of the workload has continuously been processed from the beginning and when the workload has been processed entirely. As shown in the figure, there is almost no difference in the generated MRCs even though 75% of the `Cluster18` workload has not been processed. The MAE between the two plotted lines is 0.067%. These types of workloads lead to very favorable outcomes for PR-MRC because the algorithm can ignore a large portion of accesses while maintaining excellent MRC accuracy due to the lack of change in the exact MRC throughout the workload.

A portion of the SEC workloads exhibit very high MAE of over 20%, contributing to the average MAE across all SEC workloads of 13%. This is due to the specific criteria we have selected in our "best" configuration. Using the *approximate miss rate* criteria, PR-MRC fails to recognize a change in the access pattern and instead stays in the *Access Pattern Monitoring Phase* for longer than it should, causing a large error to accumulate in the MRC. If we were to switch our criteria from *approximate miss rate* to *working set size*, all SEC workloads would result in a MAE of less than 2%, with an average MAE of 0.26%.

Cluster25 is the only Twitter workload to have worse MAE using PR-MRC when compared to SHARDS, where PR-MRC has a MAE of 0.0157% and SHARDS at a sampling rate of R = 0.1 has a MAE of 0.0002%. However, in both cases, the MAE is considered to be very good (we do not show this result on a graph as the miss rate drops to below 0.05% immediately for all methods of MRC generation).

Regardless of our selection of the "best" configuration for PR-MRC, each of the three criteria used for the *Access Pattern Monitoring Phase* produce different MRC results that can be compared to the exact MRC using Olken. Figure 4.9 shows the generated MRCs for `Twitter Cluster44` using all three criteria and the lowest v_{th} parameter value for each PR-MRC configuration. For this workload, the *working set size* criteria generates a more accurate MRC then the other two criteria, however in other workloads a different criteria may have better accuracy.

Figure 4.10 shows the MAE for all tested workloads across the MSR, SEC, and Twitter sets when using SHARDS at a sampling rate of R = 0.1. Comparing against the MAE for all tested workloads when using PR-MRC in Figure 4.7, SHARDS has worse accuracy on the MSR workloads

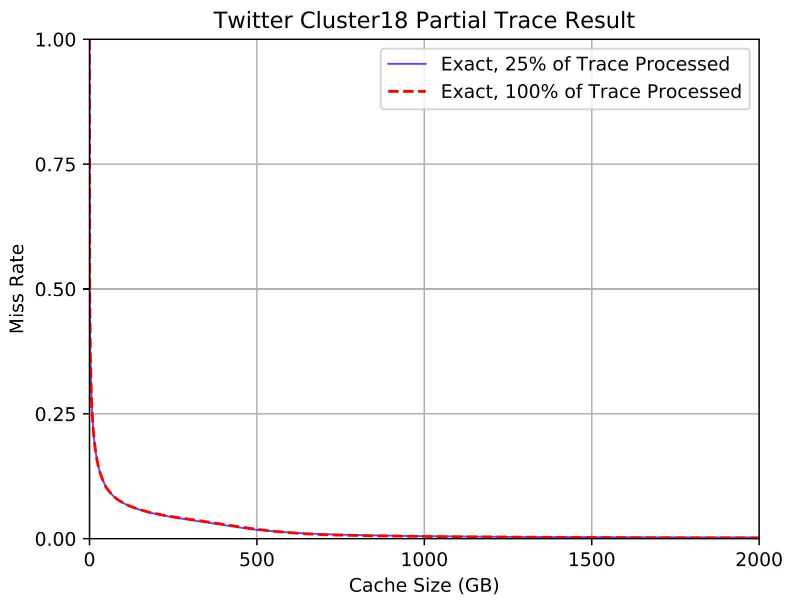

Figure 4.8: Miss Ratio Comparison between 25% of the workload processed and 100% of the workload processed for the `Cluster18` workload using rounded next power of two block sizes.

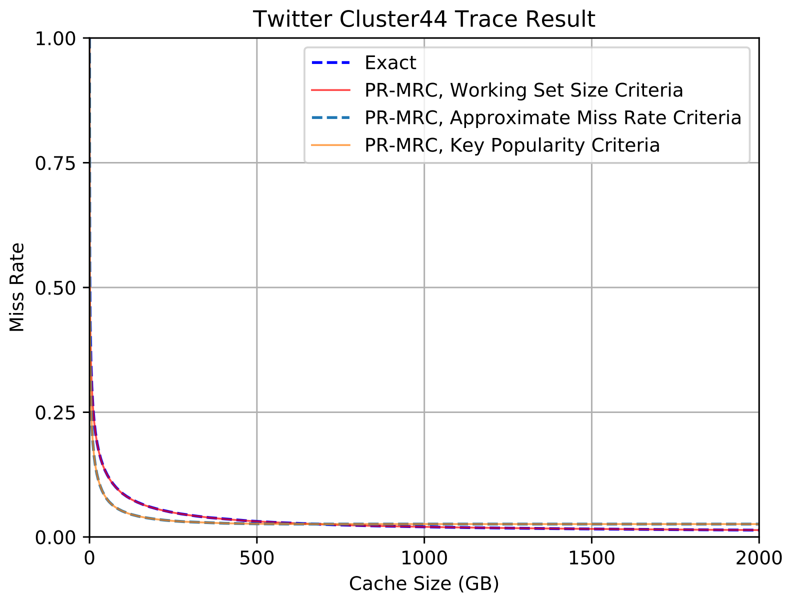

Figure 4.9: MRC Comparison between the exact MRC and PR-MRC when using each criteria independently for the `Twitter Cluster44` workload.

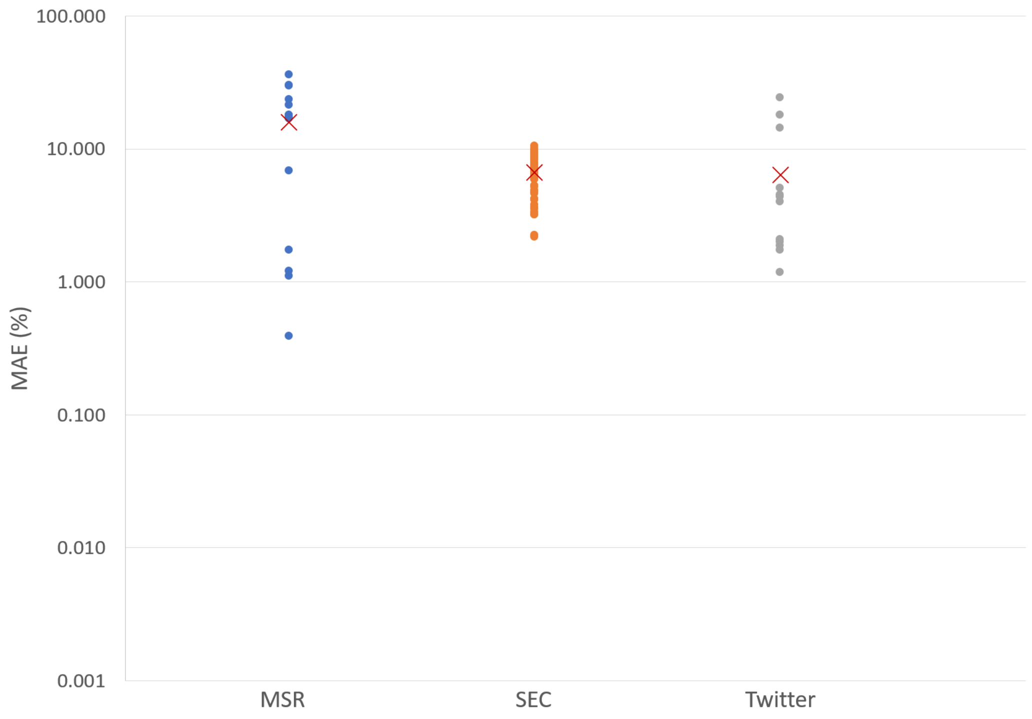

Figure 4.10: MAE of SHARDS R = 0.1 against Olken across all tested workloads, including the average MAE for each configuration shown as a red X. Note the logarithmic scale of the y-axis.

and the Twitter workloads, but has better accuracy on the SEC workloads. However, by switching our criteria from *approximate miss rate* to *working set size*, we show that PR-MRC has better accuracy results on the SEC workloads than SHARDS. This shows that, on average and dependent on the *Access Pattern Monitoring Phase* criteria, PR-MRC has improved MRC accuracy compared to SHARDS at a sampling rate of R = 0.1.

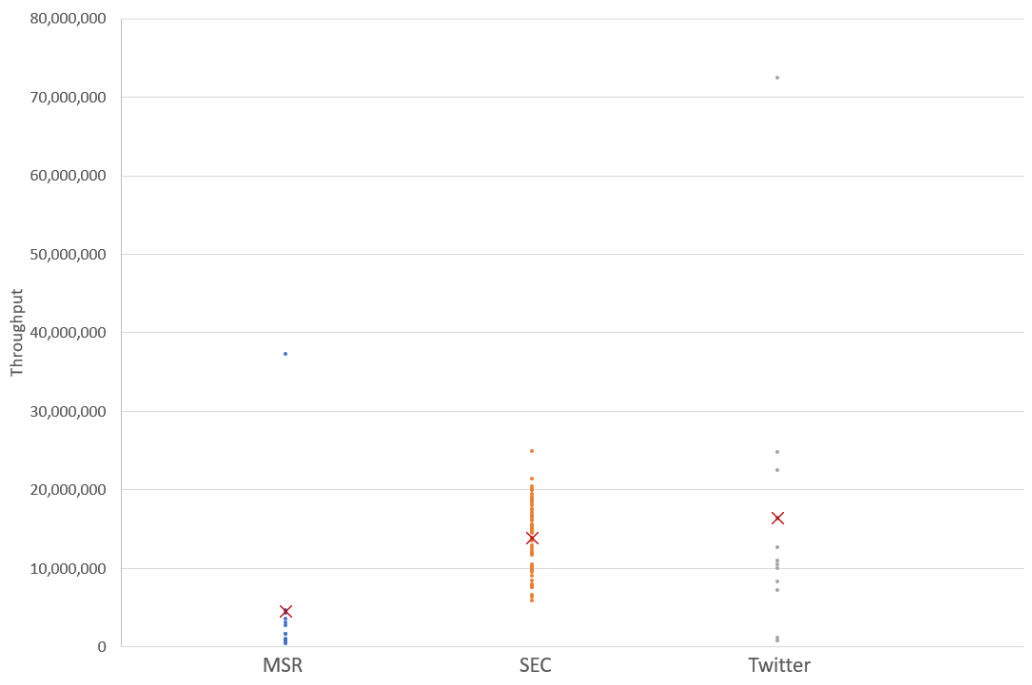

Figure 4.11: Throughput of PR-MRC across all tested workloads, including the average throughput for each configuration shown as a red X.

4.4 Throughput of PR-MRC

We compare the throughput of the PR-MRC generated MRCs using the "best" configuration and SHARDS generated MRCs. As stated previously, I/O time is omitted from throughput calculations to ensure fairness since some traces were input from RAM disks while others were not. The throughput is calculated as the number of accesses processed per second, which is obtained by dividing the total number of accesses processed with the total time taken by the algorithm. Table 4.4 shows the average throughput for each set of traces (MSR, SEC, and Twitter) for the exact Olken algorithm, PR-MRC, and SHARDS at a sampling rate of R = 0.1 and R = 0.01, when using the same parameters and criteria as used in the accuracy discussion. To account for the fact that the SHARDS algorithm reduces the total number of accesses taken into consideration, we calculate the throughput based on the original total number of accesses before sampling is applied. Figure 4.11 also shows the average throughput for PR-MRC while including the throughput for each tested workload.

Based on the table, we can determine that SHARDS requires the least amount of time to generate an MRC. One thing to note when looking at the throughput of the SHARDS algorithm is that for a given sampling rate of R = 0.1, the expected throughput should be 10x that of Olken since it only

Traces	traces	Exact (Olken)	PR-MRC	SHARDS R = 0.1	SHARDS R = 0.01
MSR Cambridge	13	3,332,665	4,839,746	10,742,932	17,027,101
SEC EDGAR	58	1,873,729	13,846,435	27,983,624	32,641,097
Twitter	13	1,454,827	16,472,193	17,426,684	39,739,649

Table 4.4: Average Throughput Comparison between Exact, PR-MRC, Shards R = 0.1, and SHARDS R = 0.01 for MSR, SEC, and Twitter workloads.

accesses 10% of the total accesses. However, in practice it is far lower than that. The SHARDS paper [27] mentions that the reason why the throughput gain can be lower than implied by the sampling rate R is because fixed overheads dominate for traces with relatively small numbers of unique accesses.

PR-MRC shows large improvements in throughput compared to the exact MRC generation for the SEC EDGAR and Twitter workloads. The MSR Cambridge workloads show a smaller improvement in throughput. This is because the workloads have considerably fewer total accesses compared to the other workloads, resulting in a small gain for PR-MRC. The throughput values for the SEC EDGAR workloads are misleading because some of the MAE's that are obtained using the "best" configuration are very high, in some cases over 30%. The high throughput is a direct result of the poor accuracy because PR-MRC stays in the *Access Pattern Monitoring Phase* for extended periods of time when in fact the access pattern is constantly changing, resulting in a high MAE.

Overall, the PR-MRC throughput for the Twitter workloads are higher than that for the other two sets of workloads. They are only 5.5% lower than the throughput of SHARDS when using a sampling rate of R = 0.1. This is because PR-MRC stays in the Access Pattern Monitoring Phase for a large portion of the overall accesses. For example, when running PR-MRC on the `Twitter Cluster25` workload, 10,231 intervals stay in the *Access Pattern Monitoring Phase*. Given that there are 11.45 billion accesses in the `Twitter Cluster25` workload, over 89% of the total accesses do not affect the LRU stack or influence the MRC generation, improving the throughput by more than 10 times the throughput of the exact algorithm (1,476,659 accesses per second compared to 31,565,309 accesses per second).

For some of the tested workloads the throughput when using PR-MRC is worse than the throughout of Olken. One example of this is the `Twitter Cluster52` workload when using the *working set size* criteria. When using PR-MRC, `Twitter Cluster52` enters the *Access Pattern Monitoring Phase* 3,850 times. This means 3.85 billion of the 11.6 billion total accesses of the workload do not update the generated MRC. However, `Twitter Cluster52` does not stay in the Access Pattern Monitoring Phase for more than one interval, instead returning to the *Convergence Phase* immediately after one interval in the *Access Pattern Monitoring Phase*. As a result, the LRU stack has to be updated for every processed access, adding overhead at the end of every *Access Pattern Monitoring Phase* interval. The end result is a worse throughput compared to the throughput when using Olken.

We can also compare the throughput of PR-MRC across the different criteria used for the *Access Pattern Monitoring Phase*. Table 4.5 shows the average throughput across the 13 Twitter traces when PR-MRC uses each of the three criteria discussed in Chapter 3.3.2 using the configuration for each criteria that achieves the highest throughput. The interval size parameter is set to $I_{\text{MP}} = 100\text{K}$ accesses for all tested criteria. The throughput of Olken and SHARDS with a sampling rate of R = 0.1 is also included for comparison.

Similar to Figure 4.2 from Chapter 4.2, the highest average throughput is achieved when using the *approximate miss rate* criteria, regardless of the interval size parameter. When PR-MRC uses the *working set size* criteria, it ignores an average of 55.5% of the total number of accesses across all 13 Twitter traces when generating the MRC. This number is in comparison to 78.7% of the total number of accesses for the *key popularity* criteria and 82.9% of the total number of accesses for the *approximate miss rate* criteria, which explains the difference in average throughput between the three criteria.

One note of mention is that none of the three criteria have the highest percentage of ignored accesses across each of the 13 Twitter traces. For example, the *key popularity* criteria results in the best throughput across all three criteria for `Twitter Cluster29`, but has the worst throughput for `Twitter Cluster6`. Inversely, the *approximate miss rate* criteria has the best throughput for `Twitter Cluster6` and is tied for the worst throughput for `Twitter Cluster29` with the *working set size* criteria. This leads to an open ended question of whether it is possible to select the "best" criteria on a per-basis trace before processing the trace.

4.5 Memory Usage of PR-MRC

We compare the memory usage between the PR-MRC generated MRCs using the same configurations as Table 4.5 and SHARDS generated MRCs. The memory usage is calculated for each set of traces; MSR, SEC EDGAR, and Twitter. We first discuss the memory usage for the *Convergence Phase*

Algorithm	Average Throughput
Olken	1,454,827
PR-MRC *working set size* v_{th} = *10%*	7,851,446
PR-MRC *approximate miss rate* v_{th} = *5%*	16,472,193
PR-MRC *key popularity* v_{th} = *5. n* = *10*	12,060,770
SHARDS *R* = *0.1*	17,426,684

Table 4.5: Average Throughput (accesses per second) across Twitter traces using different PR-MRC criteria during the Access Pattern Monitoring Phase.

and then the memory usage for the *Access Pattern Monitoring Phase*.

In the *Convergence Phase*, each access is processed using a traditional, exact MRC generation algorithm. Since we are using Olken as our exact MRC generation algorithm, the memory usage for the *Convergence Phase* is O(M) where M is equal to the number of uniquely accessed keys. An additional layer of memory usage is added for the *Convergence Phase* to store the stack distance histogram of the previous interval used to generate the previous interval's MRC. Given that we only compare points on the MRC up to a maximum cache size of 2TB, the number of buckets in the stack distance histogram are fixed. Therefore, the memory usage of the additional stack distance histogram is constant.

The memory usage for the *Access Pattern Monitoring Phase* is dependent on the criteria we use to determine whether or not the access pattern has changed. As stated in Section 4.2, the *working set size* criteria has a memory overhead of O(M) where M is equal to the number of uniquely accessed keys being seen for the first time in the interval. The *approximate miss rate* criteria has a memory overhead of O(M) where M is equal to the number of uniquely accessed keys being seen for the first time in the interval that fit in the simulated non-evicting small cache of fixed size. Lastly, the *key popularity* criteria has a memory overhead of O(M) where M is the number of uniquely accessed keys in the interval.

Table 4.6 shows the maximum memory usage for the three different criteria when running each set of MSR, SEC EDGAR, and Twitter traces. The results show that the improvement in memory

Algorithm	MSR Cambridge *13 traces*	SEC EDGAR *58 traces*	Twitter *13 traces*
Olken	13,276 MB	1,089 MB	0.55 MB
PR-MRC *working set size* v_{th} = *10%*	10,101 MB	1,823 MB	0.36 MB
PR-MRC *approximate miss rate* v_{th} = *10%*	17,250 MB	964 MB	0.34 MB
PR-MRC *key popularity* v_{th} = *5. n* = *10*	68,951 MB	1,070 MB	0.48 MB
SHARDS *R* = *0.1*	1,733 MB	152 MB	0.07 MB

Table 4.6: Memory Usage across all MSR, SEC, and Twitter traces using different PR-MRC criteria during the Access Pattern Monitoring Phase.

usage is mixed depending on the criteria used and the set of traces tested on. However, all tested configurations of PR-MRC have larger memory usage than SHARDS at a sampling rate of R = 0.1. The Twitter traces exhibit the best improvement in memory usage regardless of the criteria used. This is because, for the Twitter traces, PR-MRC ignores the largest average percentage of accesses when updating the MRC. This reduces the memory required for the AVL tree data structure used to hold the uniquely accessed keys.

The memory usage for the *key popularity* criteria is worse than Olken for the MSR traces, but better for the SEC EDGAR traces. This is due to the fact that the *key popularity* criteria needs to keep a hashtable of all uniquely accessed keys in the interval. The SEC EDGAR traces have a very small number of uniquely accessed keys in comparison to the MSR traces, which is the reason why the MSR traces result in a large memory usage for the *key popularity* criteria.

The *working set size* criteria when tested on the SEC EDGAR traces show a large increase in memory usage. This is because the majority of SEC EDGAR traces do not stay in the *Access Pattern Monitoring Phase* for a large portion of the total accesses. This means the AVL tree data structure is updated for the majority of the trace resulting in a similar memory usage to the AVL tree data structure used in Olken. The additional memory usage factors in from the constant memory usage to hold the previous interval's stack distance histogram in the *Convergence Phase* and the memory usage for running PR-MRC in the *Access Pattern Monitoring Phase*.

While we show in Table 4.6 that the memory usage of PR-MRC can be lower than that of

Olken using certain configurations, the memory usage of PR-MRC will never be lower than that of SHARDS at a sampling rate of R = 0.1. This is because SHARDS performs its sampling based on the keys of the accesses, guaranteeing that only a portion of the total keys will be accessed, thus reducing the amount of memory required for the AVL tree data structure. PR-MRC performs its sampling across fixed size intervals of the accesses which does not guarantee that a portion of the total number of unique keys will be accessed. This leads to scenarios where PR-MRC stays in the *Access Pattern Monitoring Phase* for the majority of the workload, ignoring a large percentage of the total accesses, while still accessing a large portion of the total number of unique keys, resulting in an AVL tree that has a similar size to an AVL tree produced by an exact MRC generation algorithm like Olken.

Chapter 5

Concluding Remarks

Miss rate curves (MRCs) are an important tool for determining the optimal sizes of caches and for dynamically resizing caches to adapt to changes in the workload. Our study of numerous MRC generation algorithms has allowed us to understand the advantages and disadvantages of applying sampling as a means of generating an approximate MRC in an effort to improve performance and reduce memory overhead of generating MRCs.

We introduced PR-MRC, a new approximate MRC generation algorithm, that (i) uses non-statistical sampling instead of the more commonly used statistical sampling, and (ii) offers trade-offs between accuracy and throughput when compared to current approximate MRC generation algorithms. Unfortunately, PR-MRC fails to provide any form of benefit on workloads where the access pattern is shown to be constantly changing, operating as Olken would with additional overhead. Under the ideal scenario where the access pattern does not change, the memory overhead is reduced when compared to Olken but worse when compared to other approximate MRC generation algorithms.

The results of PR-MRC are shown to be worse than initially hypothesized, and even worse after comparing with the adjusted version of SHARDS, $SHARDS_{adj}$, which we were not aware of at the time of our experimental work. As introduced in the experimental evaluation section of the SHARDS paper [27]; for cases where SHARDS introduces non-trivial error in the MRC, $SHARDS_{adj}$ applies a vertical shift for the sections with difference while the finer features of the MRC are modeled accurately. As a result, the accuracy of the MRCs are improved significantly at virtually no cost.[1] This makes the trade-offs of PR-MRC less ideal since the $SHARDS_{adj}$ algorithm produces an acceptable MAE while having better throughput than PR-MRC.

The main focus of any future work into PR-MRC would explore different phase change criteria. The different phase change criteria we considered for the Access Pattern Monitoring Phase showed us that there are many different criteria that can be applied. No criteria that we used was the perfect criteria across all tested workloads and for some of them, the criteria incorrectly assumed that the access pattern had not changed for the given workload. Further, exploring different phase change criteria may lead to a more optimal criteria. Similar to the work done on low cost working set size tracking [32] that not only uses the working set size, but data TLB misses, L1 cache misses, and L2 cache misses to infer whether a phase change has occurred or not, PR-MRC could also use a combination of phase change criteria to infer phase changes.

Other future work involves testing across a wider variety of workloads. One of these workloads that we did not have the time to test are the IBM workloads. These workloads are categorized as disk storage workloads, which are different from the majority of traces we have looked at and experimented on that are categorized as web traffic workloads. The differences between the two types of workloads and the MRCs that they generate could give us an insight as to why PR-MRC performs poorly on some workloads and not others.

Given the numerous phase change criteria that are available and the number of parameters used for both phases of PR-MRC, it may be possible to apply machine learning so that PR-MRC can apply the set of parameters and criteria that offer the best results at the current processing state of the target workload, allowing the algorithm to adapt in real time. This can lead to a small advantage over other currently existing approximate MRC generation algorithms because PR-MRC specifically makes its decision based on prior information of the workload that has already been processed.

As far as we have seen in the literature, no approximate MRC algorithm is perfect. Out of the MRC generation algorithms we have studied, all but $SHARDS_{adj}$ have shown errors higher than expected for some number of existing workloads, while $SHARDS_{adj}$ has not been extensively tested and published. This shows that we need to better understand why this occurs and improve the algorithms accordingly. One of the biggest takeaways is that MRCs are useful in practice only if

[1] A proper evaluation of $SHARDS_{adj}$ has so far not been published; the original SHARDS paper evaluated $SHARDS_{adj}$ on a small subset of their total workloads.

they can be trusted. If you cannot trust the MRC, then making a decision on how to provision the size of the cache based on the MRC is irrelevant, regardless of what MRC generation algorithm is used.

Bibliography

[1] *As a team that greatly benefits from open-source software, these are the projects that we have contributed back to the community.* URL: http://shopify.github.io/.

[2] *Audit sampling.* 2021. URL: https://corporatefinanceinstitute.com/resources/knowledge/accounting/what-is-audit-sampling/.

[3] Giuseppe DeCandia et al. "Dynamo: Amazon's highly available key-value store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (2007), pp. 205–220. ISSN: 0163-5980. DOI: 10.1145/1323293.1294281. URL: https://doi.org/10.1145/1323293.1294281.

[4] Ashutosh S Dhodapkar and James E Smith. "Comparing program phase detection techniques". In: *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-03), 2003.* IEEE. 2003, pp. 217–227.

[5] Zachary Drudi. "A streaming algorithms approach to approximating hit rate curves". PhD Thesis. University of British Columbia, 2014.

[6] *Edgar Log Data Sets.* 2015. URL: https://www.sec.gov/dera/data/edgar-log-file-data-set.html.

[7] Randal J Elder et al. "Audit sampling research: A synthesis and implications for future research". In: *Auditing: A Journal of Practice & Theory* 32.Supplement 1 (2013), pp. 99–129.

[8] Brad Fitzpatrick. "Distributed caching with Memcached". In: *Linux J.* 2004.124 (2004), pp. 1–5. ISSN: 1075-3583.

[9] Philippe Flajolet et al. "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm". In: *Proceedings of the 2007 International Conference on Analysis of Algorithms (AOFA '07)*, pp. 127–146.

[10] Stefan Heule, Marc Nunkesser, and Alexander Hall. "HyperLogLog in Practice: Algorithmic engineering of a state of the art cardinality estimation algorithm". In: *Proceedings of the 16th International Conference on Extending Database Technology*. (EDBT '13). Genoa, Italy: Association for Computing Machinery, 2013, pp. 683–692. ISBN: 9781450315975. DOI: 10.1145/2452376.2452456. URL: https://doi.org/10.1145/2452376.2452456.

[11] Joel Hruska. *Why RAM Prices Are Through the Roof*. 2018. URL: https://www.extremetech.com/computing/263031-ram-prices-roof-stuck-way.

[12] Xiameng Hu et al. "Kinetic modeling of data eviction in cache". In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, June 2016, pp. 351–364. ISBN: 978-1-931971-30-0. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/hu.

[13] Jung-Hoon Lee, Gi-Ho Park, and Shin-Dug Kim. "A new NAND-type flash memory package with smart buffer system for spatial and temporal localities". In: *Journal of Systems Architecture* 51.2 (2005), pp. 111 –123. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2004.10.002. URL: http://www.sciencedirect.com/science/article/pii/S1383762104001122.

[14] Keqin Li and Kam Hoi Cheng. "A two dimensional buddy system for dynamic resource allocation in a partitionable mesh connected system". In: *Proceedings of the 1990 ACM Annual Conference on Cooperation*. 1990, pp. 22–27.

[15] R. L. Mattson et al. "Evaluation techniques for storage hierarchies". In: *IBM Systems Journal* 9.2 (1970), pp. 78–117. DOI: 10.1147/sj.92.0078.

[16] Dushyanth Narayanan, Austin Donnelly, and Antony Rowstron. "Write Off-Loading: Practical power management for enterprise storage". In: *ACM Trans. Storage* 4.3 (Nov. 2008), pp. 1–23. ISSN: 1553-3077. DOI: 10.1145/1416944.1416949. URL: https://doi.org/10.1145/1416944.1416949.

[17] Rajesh Nishtala et al. "Scaling Memcache at Facebook". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, Apr. 2013, pp. 385–398. ISBN: 978-1-931971-00-3. URL: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala.

[18] F. Olken. “Efficient methods for calculating the success function of fixed-space replacement policies”. In: (). DOI: 10.2172/6051879. URL: https://www.osti.gov/biblio/6051879.

[19] Cheng Pan et al. “PACE: Penalty aware cache modeling with enhanced AET”. In: *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 2018, pp. 1–8.

[20] *Pricing – Azure cache for Redis: Microsoft Azure*. URL: https://azure.microsoft.com/en-ca/pricing/details/cache/.

[21] redislabs. URL: https://redis.io/.

[22] ROBERT H. RIFFENBURGH. “Chapter 26 - Methods you might meet, but not every day”. In: *Statistics in Medicine (Second Edition)*. Ed. by Robert H. Riffenburgh. Second Edition. Burlington: Academic Press, 2006, pp. 521–529. ISBN: 978-0-12-088770-5. DOI: https://doi.org/10.1016/B978-012088770-5/50066-6. URL: https://www.sciencedirect.com/science/article/pii/B9780120887705500666.

[23] James Ryans. *Using the EDGAR log file data set*. 2017. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2913612.

[24] Trausti Saemundsson et al. “Dynamic performance profiling of cloud caches”. In: *Proceedings of the ACM Symposium on Cloud Computing*. 2014, pp. 1–14.

[25] David K. Tam et al. “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*. ASPLOS XIV. Washington, DC, USA: Association for Computing Machinery, 2009, pp. 121–132. ISBN: 9781605584065. DOI: 10.1145/1508244.1508259. URL: https://doi.org/10.1145/1508244.1508259.

[26] Carl Waldspurger et al. “Cache modeling and optimization using miniature simulations”. In: *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. 2017, pp. 487–498.

[27] Carl A. Waldspurger et al. “Efficient MRC construction with SHARDS”. In: *13th USENIX Conference on File and Storage Technologies (FAST 15)*. Santa Clara, CA: USENIX Association, Feb. 2015, pp. 95–110. ISBN: 978-1-931971-201. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger.

[28] Jake Wires et al. “Characterizing storage workloads with counter stacks”. In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 335–349. ISBN: 978-1-931971-16-4. URL: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires.

[29] Xiaoya Xiang et al. “Linear-time modeling of program working set in shared cache”. In: *2011 International Conference on Parallel Architectures and Compilation Techniques*. IEEE. 2011, pp. 350–360.

[30] Juncheng Yang, Yao Yue, and KV Rashmi. “A large scale analysis of hundreds of in-memory cache clusters at Twitter”. In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 2020, pp. 191–208.

[31] Weiming Zhao et al. “Efficient LRU-based working set size tracking”. In: *Michigan Technological University Computer Science Technical Report* (2011).

[32] Weiming Zhao et al. “Low cost working set size tracking”. In: *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. 2011.

[33] Yutao Zhong and Wentao Chang. “Sampling-based program locality approximation”. In: *Proceedings of the 7th International Symposium on Memory Management*. ISMM ’08. Tucson, AZ, USA: Association for Computing Machinery, 2008, 91–100. ISBN: 9781605581347. DOI: 10.1145/1375634.1375648. URL: https://doi.org/10.1145/1375634.1375648.