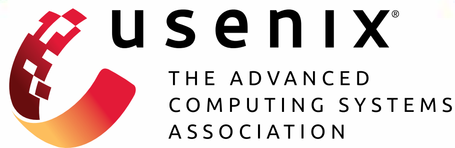

Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation

Kia Shakiba, Sari Sultan, and Michael Stumm, *University of Toronto*

https://www.usenix.org/conference/fast24/presentation/shakiba

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

NetApp®

Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation

Kia Shakiba, Sari Sultan, and Michael Stumm
University of Toronto

Abstract

In-memory caches play an important role in reducing the load on backend storage servers for many workloads. Miss ratio curves (MRCs) are an important tool for configuring these caches with respect to cache size and eviction policy. MRCs provide insight into the trade-off between cache size (and thus costs) and miss ratio for a specific eviction policy. Over the years, many MRC-generation algorithms have been developed. However, to date, only Miniature Simulations is capable of efficiently generating MRCs for popular eviction policies, such as *Least Frequently Used* (LFU), *First-In-First-Out* (FIFO), 2Q, and *Least Recently/Frequently Used* (LRFU), that do not adhere to the inclusion property. One critical downside of Miniature Simulations is that it incurs significant memory overhead, precluding its use for online cache analysis at runtime in many cases.

In this paper, we introduce Kosmo, an MRC generation algorithm that allows for the simultaneous generation of MRCs for a variety of eviction policies that do not adhere to the inclusion property. We evaluate Kosmo using 52 publicly-accessible cache access traces with a total of roughly 126 billion accesses. Compared to Miniature Simulations configured with 100 simulated caches, Kosmo has lower memory overhead by a factor of 3.6 on average, and as high as 36, and a higher throughput by a factor of 1.3 making it far more suitable for online MRC generation.

1 Introduction

In-memory caches play an important role in reducing the load on backend storage servers for many workloads [1–6]. These caches improve scalability and can reduce the latency of data access requests by serving data directly from main memory. Redis [7] and Memcached [8] are two popular in-memory caches, both of which are open source and often provided as a service by cloud providers [9–12].

In-memory caches can consume a large portion of a data center's operating budget, sometimes exceeding 60% of the total operating cost [13]. In cloud-hosted environments, such caches are priced proportionately to their size. As such, it is important to provision each cache to the "right" size using the cost-performance trade-offs for its workloads: caches that are too small incur higher miss ratios and thus higher backend storage server loads, while caches that are too large consume unnecessary resources and have higher operational costs.

One of the most effective tools to understand the trade-off between cache size and miss ratio is the *miss ratio curve* (MRC), and over the years, many MRC-generation algorithms have been developed [14–22]. An MRC plots a cache's miss ratio as a function of the cache size. Figure 1 depicts an example of such an MRC. The MRC shows the effect on the miss ratio of varying the cache's size from 0GiB to 400GiB under the MSR `src1` workload [23] using the *Least Frequently Used* (LFU) eviction policy. There is a sudden drop in the miss ratio between roughly 160GiB and 190GiB. Such a drop is referred to as a cliff and knowledge of its presence is particularly useful: if the cache were initially configured with 160GiB of memory, the MRC indicates that increasing the cache size by 30GiB would result in roughly 30% improvement in the miss ratio. The plateaus between 70GiB and 160GiB, and 190GiB and 270GiB are also informative: they indicate that if the cache is currently configured to a size within one of the plateaus, then the size can be decreased to 70GiB or 190GiB, respectively, without severely impacting the miss ratio.

The choice of eviction policy is also an important factor in configuring an in-memory cache. While most in-memory caches default to the *Least Recently Used* (LRU) eviction policy, it has been shown that under certain workloads, caches operate more efficiently using non-LRU eviction policies [3, 24–27]. For example, the LFU eviction policy can sometimes achieve a roughly 14% reduction in miss ratio when allocated the same cache size and under the same workload [27]. Eviction policies such as *First-In-First-Out* (FIFO) also have lower computational and memory overheads than other policies, such as LRU [3].

To optimize a cache configuration in terms of both size and eviction policy, it is necessary to generate an MRC for

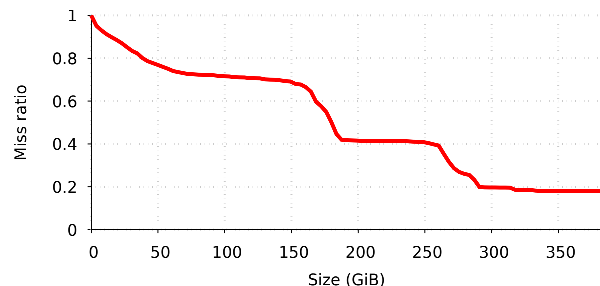

Figure 1: *MRC generated for MSR `src1` workload [23] using the Least Frequently Used (LFU) eviction policy.*

each eviction policy under consideration. However, a key limitation of almost all existing MRC-generation algorithms is that they only model caches operating with an eviction policy that satisfies the inclusion property (e.g., LRU); as such, they do not support eviction policies such as LFU, FIFO, 2Q, *Least Recently/Frequently Used* (LRFU), or *Most Recently Used* (MRU). The only known MRC generation algorithm capable of modeling a wide array of non-LRU caches with reasonable computational efficiency is Miniature Simulations (*MiniSim*) [18]. It runs individual simulations of caches of different sizes and makes use of the SHARDS [17] sampling algorithm to improve its runtime performance. However, MiniSim has several serious drawbacks, the most notable being its high memory usage. MiniSim effectively simulates independent caches of varying sizes, often causing duplicate data to be stored in the internal structures of many of these simulated caches. We found through experimentation with numerous workloads that MiniSim configured with 100 simulated caches consumes the following amounts of memory on average: 113MiB for the LFU eviction policy, 57MiB for FIFO, 40MiB for 2Q, and 31MiB for LRFU, with up to 3.1GiB for the LFU eviction policy, 1.72GiB for FIFO, 396MiB for 2Q, and 597MiB for LRFU in extreme cases. Further, to generate MRCs for multiple eviction policies simultaneously, these memory requirements are compounded. With these memory requirements, MiniSim will likely consume substantial memory, and hence may even interfere with the cache itself.

This paper introduces **Kosmo**, an MRC generation algorithm that supports the simultaneous generation of MRCs for a variety of eviction policies while, on average, using significantly less memory than MiniSim, making it better suited for online MRC generation. Kosmo uses a novel method of calculating reuse distances through the introduction of *eviction maps*. We show how Kosmo can be used to simultaneously generate MRCs for six eviction policies: LFU, FIFO, 2Q, LRFU, LRU, and MRU. Notably, LFU, FIFO, 2Q, LRFU, and MRU do not adhere to the inclusion property (§2.2).

We evaluate Kosmo using a total of 52 publicly-available workloads and measure memory usage, throughput, and accuracy for LFU and FIFO, and 33 workloads for the 2Q and LRFU eviction policies. Kosmo requires an average of 3.6 times less memory, and up to 36 times less than MiniSim across all eviction policies. Kosmo has an average throughput 1.3 times that of MiniSim across all eviction policies. Finally, Kosmo, which is also an approximate generation algorithm, produces MRCs with comparable accuracy to those generated by MiniSim.

Contributions. The contributions we make in this paper are:

- We introduce Kosmo, a novel method of simultaneously generating MRCs for a variety of eviction policies.
- We introduce a method of reconstructing the stacks of caches of varying sizes using a single copy of the cached data through our novel data structure, *eviction maps*.
- We describe how to apply eviction maps to the LFU, FIFO, 2Q, LRFU, LRU, and MRU eviction policies, allowing Kosmo to generate MRCs for these policies.
- We evaluate the performance of both Kosmo and MiniSim and show that Kosmo achieves an average memory reduction of a factor of 3.6 and up to a factor of 36.
- We examine to what degree different eviction policies violate the inclusion property.

Limitations. The work we present has several limitations, however. First, we only describe Kosmo for six sample eviction policies. Although we know Kosmo supports additional eviction policies beyond those described in this paper, it remains an open problem which classes of eviction policies Kosmo is able to support. Second, the MRCs Kosmo generates are monotonically decreasing which could increase the error for eviction policies which display significant non-monotonic behaviour. Finally, we recognize that the performance of MiniSim is affected by the performance of the underlying cache it is simulating.

2 Background

In this section, we discuss relevant prior work. We first describe the eviction policies that are the focus of this paper, namely: LFU, FIFO, 2Q, LRFU, LRU, and MRU. We then discuss the inclusion property and its importance in MRC generation. Next, we describe several key MRC generation algorithms which provide the necessary background to understand the Kosmo algorithm. Mattson's algorithm gives insight into how MRC generation can be done for policies that do not violate the inclusion property (generally referred to as "stack-based eviction policies"). SHARDS is the sampling algorithm used by both Kosmo and MiniSim. MiniSim is currently the only known, reasonably computationally efficient method for generating MRCs for caches with non-stack-based eviction policies. It is the current state-of-the-art algorithm to which we compare Kosmo.

2.1 Eviction policies

LFU (Least Frequently Used) evicts the least frequently used object in the cache to make room for new objects. A simple method of implementing this policy is to use a stack of objects, ordered firstly by frequency count and secondly by last access time. If an object needs to be evicted, the one with the smallest frequency count, or oldest time on a tie, is selected. LFU caches often outperform LRU caches in workloads that exhibit a Zipfian distribution [24, 25].

FIFO (First-In-First-Out) evicts objects in the same order in which they first entered the cache. It can be implemented using a stack[1] of objects ordered by their entry times where the object with the oldest time is selected for eviction. This policy has been found to perform well on large workloads

[1] This is sometimes also referred to as a "queue" in this context. In this paper, however, we refer to the internal data structure which holds the objects of a cache as a "stack," regardless of eviction policy.

in which accesses have large inter-arrival gaps, such as those exhibited by scanning behaviours [3, 26]. Of all the eviction policies, it is the most efficient to implement [3].

LRU (Least Recently Used) and **MRU** (Most Recently Used) evict the least recently and most recently accessed object in the cache, respectively, to make room for a new object. To implement these policies, a stack of objects sorted by their last access times is used, where the object with the oldest (LRU) or youngest (MRU) time is evicted. LRU is perhaps the most widely-used eviction policy, though MRU has been found to perform better when the workload is cyclical [28].

2Q [29] maintains objects in a cache in two separate stacks: one for objects which have been accessed only once (the *A1* stack), and one for objects which have been accessed multiple times (the *Am* stack). Objects in the A1 stack are evicted in FIFO order, and objects in the Am stack are evicted in LRU order. The A1 stack is further partitioned into two stacks referred to as *A1in* and *A1out*[2] of size *Kin* and *Kout*, respectively, where Kin and Kout are ratios of the total cache size (the authors note that a Kin value of 25% and a Kout value of 50% work well in most cases). The A1in and A1out stacks differentiate themselves in the handling of objects that get accessed a second time; if the accessed object is in the A1out stack, it gets promoted to the Am stack, while if it is in the A1in stack, it does not. Upon the first access to an object, it is placed at the head of the A1in stack. If the A1in stack is full, the oldest object in the stack is removed and placed at the head of the A1out stack. If the A1out stack is full, the oldest object is evicted from the cache. If an object which already exists in the A1out stack is accessed, it is removed from the A1out stack and placed at the head of the Am stack. If the Am stack is full, an object is evicted using LRU.

LRFU (Least Recently/Frequently Used) combines objects' recency (i.e., the time since the object was last accessed) and frequency counts to determine which object to evict. Each object has an associated *Combined Recency and Frequency* (CRF) value computed as $CRF = \sum_{i=1}^{k} F(t_{now} - t_{access_i})$, where k is the number of times the object has been accessed previously, t_{now} is the current time, t_{access_i} is the time at which the object was accessed the ith time, and $F(x)$ is defined as $F(x) = (\frac{1}{p})^{\lambda * x}$, where p is a value greater than or equal to two, and λ is a value between 0 and 1. Tuning the value of λ allows the cache to behave more similarly to an LFU cache (with λ closer to 0) or an LRU cache (with λ closer to 1). The object with the smallest CRF value is selected for eviction. Although the described CRF formula requires the full history of the object's access times, the authors note that given an object's last access time and last CRF value, one can calculate the updated CRF value without needing the object's access history using $CRF_{updated} = F(0) + F(t_{now} - t_{last_access}) * CRF_{last}$. The LRFU policy has been shown to outperform many other policies for a number of important workloads [27].

[2] The A1out "ghost" stack holds references to objects, not their values.

2.2 Inclusion property

An important characteristic of an eviction policy is whether or not it adheres to the *inclusion property*. This property states that all objects that exist in a cache of size S at a given time, also exist in any cache of size $S' > S$, when given the same access trace [18, 30]. An extension of this property is the *strict inclusion property* which adds the further constraint that all common objects in any two caches (with the same access trace) must be in the same order in the caches' internal data structures (i.e., stacks).

An MRC generation algorithm that models an eviction policy adhering to the strict inclusion property is often referred to as a "stack algorithm," and can be implemented similarly to Mattson [14], described below. If the eviction policy does not adhere to the strict inclusion property, a dedicated algorithm for the eviction policy or MiniSim must be used.

In the literature, the LFU eviction policy is often referred to as a stack algorithm, which implies it can be modeled using an algorithm similar to Mattson [14, 17, 30]. However, this is only the case for so called *ideal LFU caches*, in which the cache maintains frequency counters for all objects that were ever accessed; if an object is evicted and accessed again in the future, its counter persists and is further incremented. In practice, LFU caches do not maintain the counters of evicted objects [31], as maintaining these counters would entail significant memory overhead. These *practical LFU cache* implementations remove the counter of any object being evicted from the stack, and if a previously evicted object is accessed again, a new counter is instantiated and initialized to one.

Practical LFU caches do not adhere to the inclusion property, in contrast to ideal LFU caches. Table 1 shows this for a simple access trace. Here, the objects and their associated counts in two practical LFU caches of size 3 and 4 are shown. At each time step, the frequency counter (shown alongside each object in brackets) of the accessed object is incremented by one, or initialized to one in the case of an object being accessed for the first time. The stack of each cache is ordered from least to most likely to be evicted from the cache (i.e., the object with the largest frequency counter, or most recently accessed if two objects have the same frequency counter, on the left). At time 9, it is evident that although the two caches were provided with the same access trace, object "e" exists in the cache of size 3, but not in the cache of size 4. This is a violation of the inclusion property.

An interesting question is whether it is feasible to use MRCs generated under the assumption of ideal LFU caches (which adhere to the strict inclusion property) to model the miss ratios of practical LFU caches. Figure 2 demonstrates that this is not the case. The figure depicts the MRCs for ideal and practical LFU caches for two workloads. For the MSR `src1` workload [23], the miss ratios deviate substantially for cache sizes between 190GiB and 240GiB. Similarly, for the MSR `web` workload [23], the miss ratios deviate significantly for cache sizes between 38GiB and 46GiB.

Table 1: Sample trace for LFU caches of sizes 3 and 4 demonstrating a violation of the inclusion property.

Time	Access	LFU cache size 3	LFU cache size 4
1	a	a(1)	a(1)
2	b	b(1), a(1)	b(1), a(1)
3	c	c(1), b(1), a(1)	c(1), b(1), a(1)
4	d	d(1), c(1), b(1)	d(1), c(1), b(1), a(1)
5	a	a(1), d(1), c(1)	a(2), d(1), c(1), b(1)
6	d	d(2), a(1), c(1)	d(2), a(2), c(1), b(1)
7	b	d(2), b(1), a(1)	b(2), d(2), a(2), c(1)
8	e	d(2), e(1), b(1)	b(2), d(2), a(2), e(1)
9	f	d(2), e(1), f(1)	b(2), d(2), a(2), f(1)

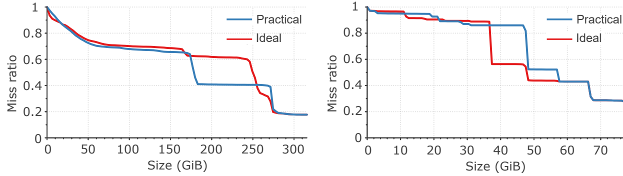

Figure 2: MRCs for ideal and practical LFU caches for the MSR `src1` (left) and `web` (right) workloads [23].

2.3 MRC generation

Mattson's algorithm. Mattson et al. were the first to describe a method capable of constructing a miss ratio curve from an LRU cache's access trace in a single pass [14]. To generate the MRC, Mattson's algorithm maintains an LRU stack of all the accessed objects. Because MRC generation algorithms simply model caches but do not store the values of the accesses, an "object" in this context refers to the referenced key or a hash of the referenced key. Upon each access to an object, if the object has been previously accessed, its *reuse distance* is measured as the number of objects ahead of it in the stack and is recorded in a histogram. The object is then moved to the front of the LRU stack. If an object has not been seen before, the reuse distance is said to be infinity and is recorded as such in the histogram; then a new object is instantiated and inserted at the front of the LRU stack. After the entire trace has been processed, the resulting MRC is generated as the inverse CDF of the histogram. The Kosmo algorithm also uses a histogram of reuse distances to generate MRCs.

Other algorithms have been introduced to improve on Mattson's computational overhead. Olken [15] maintains the objects in a balanced tree, sorted by their last access times to bring the compute complexity from $O(MN)$ to $O(NlogM)$, where M and N are the number of unique and total number of accesses, respectively. Parda [16] extends the Olken algorithm to support the parallel processing of an access trace.

SHARDS. Waldspurger et al. describe an algorithm called *SHARDS* which works in conjunction with an exact MRC generation algorithm, such as Olken. SHARDS uses Olken to generate the MRC, but uses only a sampled subset of the trace [17]. This significantly improves the efficiency with which an MRC can be generated, and while the resulting MRC is approximate, it typically has reasonably low error [17]. Kosmo also uses SHARDS to improve its performance.

SHARDS can be implemented as either fixed-rate or fixed-size. The *fixed-rate* implementation samples the trace using a specific rate, R. The authors of the paper found that an R value of 0.001 results in reasonably accurate MRCs, thus reducing the overhead by a factor of 1,000 [17]. The *fixed-size* implementation extends the fixed-rate implementation by adjusting the sampling rate downward so as to limit the number of objects that exist in the MRC algorithm's internal data structures at any given time to a constant, S_{max}. In our experimentation, we use S_{max} values of 1,024 and 2,048 when running Kosmo (as was done in the SHARDS paper).

The authors of SHARDS noticed that the expected number of sampled accesses, $E[N_S]$, was often not equal to the measured number, N_S. To correct for this, after the access trace has been fully processed, the difference between $E[N_S]$ and N_S is added to the first histogram counter (i.e., the histogram counter for the smallest reuse distance). By applying this correction, the authors achieved significant improvements in the error induced by the sampling algorithm [17].

Miniature Simulations. Waldspurger et al. describe a method of generating MRCs called *Miniature Simulations* (which we will refer to as "MiniSim"), capable of modelling any eviction policy. MiniSim independently simulates caches at varying sizes to obtain the resulting MRC [18].

To generate an MRC using MiniSim, a maximum simulated cache size, C_{max}, is selected. A number of simulated caches (N_C) are then instantiated, each simulating a cache size between C_{max}/N_C and C_{max}. In practice, N_C is often set to 100. Upon each access in a trace, the access is processed by each simulated cache. When the trace is complete, the miss ratios of the simulated caches and each simulated cache's respective size are used to form an MRC.

MiniSim utilizes the sampling method proposed by SHARDS to operate on a small subset of the total trace to improve runtime performance, making it an approximate MRC generation algorithm.

Although MiniSim can generate MRCs for any eviction policy, it has two key shortcomings. First, it has high memory usage as each data point on the curve simulates an instance of a cache, and the different simulations of the caches do not share any of their internal data structures. These caches, especially those with similar sizes, often contain many of the same objects, yet each cache allocates memory for these objects independently. Experimentally, we found that MiniSim used an average of 113MiB to generate a single MRC for the LFU eviction policy, with up to a maximum of 3.1GiB. To reduce this memory usage significantly, one would have to reduce the sampling rate of SHARDS or reduce the number of simulated caches, which in turn would reduce the accuracy of the resulting MRC.

A second key shortcoming is that the range of cache sizes to be simulated must be defined before the input trace is first processed and cannot be modified while the simulation is ongoing. This is limiting when generating MRCs online for live

workloads, where the workload's working set size is unknown ahead of time. Because the maximum simulated cache size C_{max} cannot be modified after the simulation begins, a large, worst case, value is typically selected to ensure the access trace's working set size (i.e., the cache size required to store all unique objects) is likely to be captured. This prevents MiniSim from being able to focus the sizes of the simulated caches to regions of the MRC which lie within the workload's actual working set size.[3] For example, although Twitter tends to overprovision its caches [3], of the publicly available access traces in the Twitter dataset, 25% have a working set size of less than 2GiB. If a large C_{max} value, such as 200GiB, is selected to model one of these access traces, the simulated caches are sized in increments of $200\text{GiB}/100 = 2\text{GiB}$, preventing points of interest on the MRC from being observable. This requires MiniSim to be configured with a large number of simulated caches as reducing this to a smaller value will further reduce the resulting MRC's granularity.

3 Kosmo

We now present Kosmo, an MRC generation algorithm capable of generating approximate MRCs for a variety of eviction policies simultaneously. We begin by presenting the algorithm in general terms (§3.2) and then describe several optimizations that significantly improve its efficiency (§3.3). We then describe the Kosmo algorithm for the LFU eviction policy (§3.4) specifically, followed by the required extensions to support other evictions policies (§3.5). We then show how Kosmo can be extended to support variable object sizes (§3.6) and TTLs (§3.7). Finally, we describe how Kosmo can generate MRCs for multiple eviction policies simultaneously (§3.8).

Both Kosmo and MiniSim simulate caches of different sizes to generate an MRC. The key difference is that MiniSim maintains a stack for each cache throughout the duration of the simulation, while Kosmo reconstructs the stacks dynamically, only as needed. MiniSim keeps track of the miss ratios of the different caches and constructs the MRC using these miss ratios once it has processed the entire access trace, while Kosmo uses an approach similar to Mattson: it records stack distances encountered in a histogram and, in the end, constructs the MRC from the histogram.

Further, MiniSim always simulates the same pre-configured cache sizes, regardless of the working set size of the access trace, while Kosmo simulates a different set of cache sizes on each access, the largest simulated cache size being the reuse distance of the currently accessed object minus one. This allows Kosmo to generate MRCs with similar error rates to that of MiniSim while simulating far fewer caches, which leads to lower memory and compute overheads for Kosmo.

The simulated caches in an instance of MiniSim do not share any internal data structures, therefore an object may exist simultaneously in the stacks of multiple caches, causing MiniSim to consume large amounts of memory. In contrast, Kosmo maintains the data representing an object only once in a global data structure. Each object in this data structure contains the minimal amount of data required to allow the stack of a cache of any size to be reconstructed dynamically.

3.1 Kosmo data structures

Kosmo maintains all objects ever accessed in a data structure called the *global table*, implemented as a dynamic hash table. Each object in the global table has an associated *eviction map* which, in turn, consists of a set of *eviction records*. Whenever an object is evicted from any of the caches (of different sizes) being simulated, an eviction record is added to the eviction map of the object. This eviction record includes or registers the size of the cache from which the object was evicted, as well as other policy-specific information described further below. Using an object's eviction map, Kosmo is able to determine at any time whether the object exists in a cache of a specified size. If it exists in the cache, Kosmo can determine its position within the cache's internal data structure, referred to as the cache's *stack*, using the policy-specific information in the eviction records. An eviction map also holds a reference to the associated object, allowing it to access the object's properties, such as the last access time or ideal frequency count in the case of LFU caches.

Eviction maps are eviction policy-specific and must be implemented on a per-policy basis. However, all eviction maps support three primary operations:

1. For a given object, identify the size of the smallest cache that contains the object.
2. For its associated object, calculate the object's sorting key, given a cache size, S. The sorting key is calculated using policy-specific information in the eviction records, allowing Kosmo to properly order objects in the stack of the cache of size S it is reconstructing.
3. Insert a new eviction record.

The specific implementation of eviction maps for the LFU eviction policy is described in §3.4. The implementations for the FIFO, 2Q, LRFU, LRU, and MRU policies are described in §3.5. The implementations for the LRU and MRU policies are provided to demonstrate Kosmo's generality and are not included in the experimental analysis.

3.2 The Kosmo algorithm

We first describe a variant of the Kosmo algorithm that is highly inefficient. Optimizations that make it efficient are described in §3.3. Upon each access, the Kosmo algorithm performs the following sequence of steps:

1. Calculate the reuse distance D of the accessed object and

[3] The sizes of MiniSim's simulated caches can be configured non-uniformly [18], though this would require knowledge of either the shape of the MRC or a specific point of interest around which to cluster the sizes of the simulated caches (e.g., the current size of the production cache) before processing the access trace. Further, the shapes of some workloads' MRCs can change dramatically over time [20, 32, 33].

update the histogram counter associated with D.

2. Reconstruct the stacks of the caches of sizes $S < D$, for every possible cache size at a byte-level granularity. These are the caches that do not contain the accessed object.
3. Select an object from each reconstructed stack for eviction (to make space for the accessed object) and place a new eviction record in the eviction map of the evicted object.

Using the accessed object's key, its associated eviction map is found using the global table. The object's reuse distance D can be determined using the object's eviction map by finding the smallest sized cache in which, according to the eviction records, the object exists. If the object is not found in the global table, its reuse distance is set to infinity, as it is being accessed for the first time. The histogram counter associated with D is then incremented.

For eviction policies that do not adhere to the inclusion property, an interesting question is: what is the reuse distance of an object? For eviction policies that *do* adhere to the inclusion property, the reuse distance is clear. It is simply the minimal cache size that contains the accessed object; any larger cache will contain the object, while any smaller cache will not. For eviction policies that do *not* adhere to the inclusion property, an object may exist in a cache of size S, but not exist in some caches of size $S' > S$. Nevertheless, we argue that the size of the smallest cache containing the accessed object should be the reuse distance. There are several motivations for this choice. First, this choice allows for an important optimization to ensure eviction maps do not contain a large number of eviction records, which we describe in §3.3. Second, the choice simplifies the calculation of an object's reuse distance. Third, in our experiments, we found it is rare for an access to cause a violation of the inclusion property and maintain this violation for large ranges of cache sizes, which we show in §4.5.

Immediately after an object O is accessed, it must exist at all cache sizes. Hence, for each cache of size S in which the object does not exist when it is accessed, some object needs to be evicted to make space for O. To select an object for eviction, Kosmo reconstructs the stack of the cache by iterating through all objects in the global table. Using each object's eviction map, Kosmo determines if the object exists in the cache (of size S) and, if so, where in the cache's stack the object resides relative to other objects using the objects' sorting keys. Through this process, Kosmo reconstructs the cache's full stack. The object at the top of the reconstructed stack is selected as the object to be evicted and a new eviction record is accordingly inserted in the object's eviction map.

It is clear that Kosmo, as described, is highly inefficient. First, because the global table contains an entry for every object ever accessed, it can grow quite large. Second, to determine which objects need to be evicted from which reconstructed stacks, Kosmo must reconstruct the stack for every cache of size less than the accessed object's reuse distance. Finally, as an eviction record is inserted into an object's eviction map each time it is selected for eviction, the eviction maps may contain a large number of eviction records.

3.3 Optimizations

We describe four optimizations: cache size granularity, eviction record pruning, the use of SHARDS, and parallel stack reconstruction.

Granularity. To reduce the number of cache stacks that need to be reconstructed on each access, a granularity parameter G is introduced. This parameter limits the number of caches that need to be reconstructed on each access to a fixed number. For example, if an object O is accessed and all cache sizes less than or equal to 3GiB are found to not contain O and must therefore be reconstructed to perform the necessary evictions, with a granularity parameter of 100, 100 simulated caches in size increments of $3\text{GiB}/100 = 30\text{MiB}$ are examined.

We experimentally evaluated appropriate values of G. A higher value of G typically means a lower MAE (mean absolute error); however, this also leads to increased computational overhead. Figure 3 shows the experimental results of varying values of G and the corresponding MAEs for all workloads in the MSR dataset [23]. An interesting observation is that Kosmo can achieve a low mean MAE with even a small value of G. As evident in this figure, selecting a G value greater than 10 does not significantly reduce the mean MAE. We therefore conservatively select 10 as our value of G and use it throughout all our simulations.

Upon accessing an object with reuse distance D, Kosmo only simulates G caches of sizes $S < D$. As a result, it is able to achieve a comparable MAE while simulating significantly fewer caches than MiniSim. The accessed object is assumed to already exist in caches of sizes $S \geq D$, so they do not need to be simulated. In contrast, MiniSim simulates all (typically 100) considered cache sizes on each access, regardless of the accessed object's reuse distance.

Eviction record pruning. To reduce the number of eviction records in an object's eviction map, each time a new eviction record is added, indicating the object is being evicted from the cache of size S, all eviction records with cache size $S' < S$ are removed. In doing so, Kosmo is effectively assuming the inclusion property where an object being evicted from a cache of size S will thereafter also not exist at any cache of size $S' < S$. This may introduce inaccuracies for eviction policies which do not adhere to the inclusion property, however, we have found these inaccuracies to be negligible (§4.4).

Pruning drastically reduces the size of each object's eviction map. Figure 4 shows the effect of pruning on the average number of eviction records in objects' eviction maps throughout a typical access trace of a workload. On average, pruning reduces the average size of the eviction maps (i.e., the numbers of records it contains) by a factor of roughly 387. This drastically reduces the memory required to store the eviction records as well as the computational overhead of searching

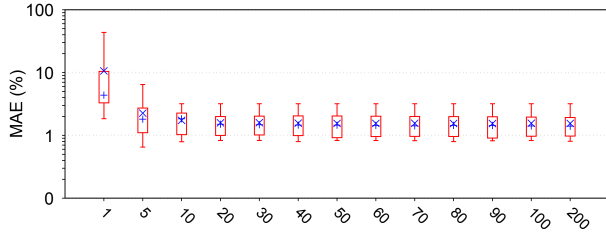

Figure 3: MAEs of varying granularities for all workloads in the MSR dataset [23]. A fixed-sized implementation of SHARDS was used with $S_{max} = 2{,}048$ for granularities between 1 and 200. The top line identifies the maximum result while the bottom line identifies the minimum. The top of the box is the 75th percentile result and the bottom of the box is the 25th percentile result. The $\times$ and $+$ symbols indicate the mean and median MAEs, respectively, for each granularity value.

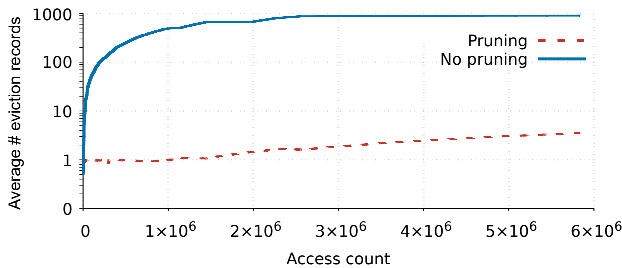

Figure 4: The average number of eviction records per object in the global table throughout the MSR web workload [23] using fixed-rate SHARDS ($R = 0.001$) shown with (dashed red line) and without (solid blue line) pruning. The unsampled access count is shown here (i.e., not the number of sampled accesses).

through the eviction records when determining if the object exists in a cache of size S.

SHARDS. On each access, Kosmo must iterate through all objects in its global table to reconstruct the stacks of various cache sizes. The global table could include billions of objects. For this reason, we use SHARDS to spatially sample the accesses. This lessens the number of objects in the global table and reduces the stack reconstruction time. The fixed-size variant of SHARDS (§2.3) is particularly useful for Kosmo as it limits the size of the global table to a known constant: the S_{max} value of SHARDS.

Parallel stack reconstruction. As the reconstruction of a cache's stack does not modify the global table, the reconstruction of the stacks of multiple caches can all be done in parallel. This improves the response time of Kosmo, but increases the memory overhead because the stacks of all caches exist in memory simultaneously. This creates a trade-off between throughput and memory usage. However, as we show in §4.4, Kosmo uses significantly less memory than MiniSim, even when reconstructing stacks in parallel.

3.4 Kosmo for LFU

To support the LFU eviction policy specifically, Kosmo maintains the following information in its data structures. First,

Algorithm 1: Eviction map for LFU cache object.

Ref : *object*
Record : *map* → *Map<cache_size, count>*
1 **Eviction Map** *LFU*:
2 **Function** *Insert(cache_size)*:
3 *map.insert(cache_size, object.global_count)*
4 **Function** *FindSmallestExisting()*:
5 **for** *record* **in** *map* **do**
6 **if** *record.count == object.global_count* **then**
7 **return** $record.size + 1$
8 **return** *object.size*
9 **Function** *GetSortingKey(cache_size)*:
10 $rec \leftarrow map.find(\geq cache_size)$
11 **if** *!rec* **then**
12 **return** *object.global_count*
13 **return** $object.global_count - rec.count$

each object in the global table maintains a timestamp of when the object was last accessed and a counter referred to as the object's *global count*. This counter is incremented by one each time the object is accessed and is therefore the same as the object's frequency count in an ideal LFU cache. Second, each eviction record in the object's associated eviction map contains the size of the cache from which the object was evicted, and the object's global count value when the eviction occurred. The latter makes it possible to infer the object's frequency count for a specific cache size, referred to as the *local count*, as it would be in a practical implementation of a cache of that size. Algorithm 1 contains the pseudocode for the eviction map's three main operations.[4]

We found that while a practical LFU cache regularly violates the strict inclusion property, it does not violate the (non-strict) inclusion property often. Although objects in the stacks of caches of different sizes may be in different orders, typically, the objects in each cache's stack are a subset of the objects in the stack of a larger cache. Experimentally, we noticed that, on average, only 1.49% of accesses to an LFU cache cause the (non-strict) inclusion property to be violated. With Kosmo, we therefore assume that the (non-strict) inclusion property holds for practical LFU caches (unlike Mattson which assumes the strict inclusion property holds) and show in our experimentation results that this produces negligible errors in the resulting MRCs.

To obtain the reuse distance of an object O when it is accessed, we search for O's eviction records to identify the record with the largest cache size S wherein the record's count value is equal to the object's current global count. This record indicates the object has not been accessed in a cache of size S since O was last evicted (at which time this record was inserted) and therefore the object does not exist in any caches

[4] Our descriptions here assume fixed-sized objects, though the algorithms shown in the listings accommodate variable-sized objects as described in §3.6 (i.e., `object.size` [variable-sized] in the listings corresponds to `1` [fixed-sized] in the text).

of sizes $S' \leq S$. The object's reuse distance is then $S+1$. If no eviction record which satisfies this condition is found, we can conclude the object exists at all cache sizes, so the reuse distance is 1.

To reconstruct the stack of a cache of size S, we use the eviction map for each object in the global table to (i) determine if the object exists in a cache of this size, and (ii) if the object exists, calculate its sorting key. To determine if the object exists in a cache of size S, we simply check if S is greater than or equal to the object's reuse distance. If the object is found to exist, its sorting key is the object's local count paired with its last access time. Once calculated, the object then can be placed in the correct position in the reconstructed stack.

To calculate the local count of an object O in a cache of size S, we search O's eviction map for an eviction record with the smallest cache size S' that satisfies $S' \geq S$. If no such record exists, then O has never been evicted from any cache of size $S' \geq S$ since O was first accessed, therefore its local count is equal to its global count. Otherwise, if an eviction record with size $S' \geq S$ is found, the local count is O's global count minus O's global count when it was evicted from the cache of size S' (i.e., the count value in the eviction record), given that the local count should equal the number of accesses to O since its last eviction.

After reconstructing the LFU stack of a cache, Kosmo selects the object at the top of the stack (i.e., the object with the smallest sorting key) for eviction. A new eviction record is inserted with the cache size and the object's global count into said object's eviction map.

3.5 Other eviction policies

The Kosmo algorithm, as described in §3.2, is not designed for any specific eviction policy. Kosmo can generate MRCs for other eviction policies by using the same process, but using policy-specific implementations for the eviction maps. Here, we describe eviction maps for five other policies, namely FIFO, 2Q, LRFU, LRU, and MRU as examples.

FIFO. The design of the FIFO eviction map is fundamentally different than that of the LFU eviction map. A FIFO eviction record indicates the cache sizes for which an object does exist in the cache, whereas an LFU eviction record indicates the cache sizes for which it does not. Each eviction record in a FIFO eviction map records the cache size S and the *entry time* of the object at size S. The record indicates an object's entry time for caches of sizes S', where $S \leq S' < S_{next}$ and S_{next} is the cache size stored in the eviction record with the next largest cache size. Algorithm 2 contains the pseudocode for the eviction map's three main operations.

Algorithm 2: Eviction map for FIFO cache object.

```
    Ref     : object
    Record  : map → Map < cache_size, timestamp >
1   Eviction Map FIFO:
2       Function Insert(cache_size):
3           rec ← map.find_largest(≤ cache_size)
4           rec.cache_size = cache_size + 1
5           map.remove(≤ cache_size)
6       Function FindSmallestExisting():
7           rec ← map.find_smallest()
8           return rec.cache_size
9       Function GetSortingKey(cache_size):
10          rec ← map.find_largest(≤ cache_size)
11          if !rec then
12              return 0
13          return rec.timestamp
```

On every access to an object, as the object must exist at all cache sizes immediately after it has been accessed, a new eviction record with a cache of size 1 is inserted into the object's associated eviction map if a record with a cache size of 1 does not already exist. This eviction record stores the entry time (i.e., the current time) of the object for any cache size smaller than the object's reuse distance (i.e., any cache size at which the object does not currently exist).

An eviction record also gets added for an object O when it gets evicted. To insert a new eviction record for a cache of size S into the eviction map of O, we first locate the record with the largest cache size S' such that $S' \leq S$. This record holds the object's entry time E into a cache of size S. We then insert a new eviction record with cache size $S+1$ and entry time E into the eviction map to indicate this object is being evicted from all caches of sizes $S' \leq S$. Finally, we perform pruning by removing all records with cache sizes $S' \leq S$.

The reuse distance of an accessed object can be determined using the object's eviction map as the smallest cache size S contained in the eviction records. By definition, a cache of size S is the smallest cache which contains the object.

When reconstructing the stack of a cache of size S, the objects' sorting keys are their entry times into the cache. To determine the entry time of an object O in a cache of size S, we find the eviction record with the largest cache size S' in O's eviction map such that $S' \leq S$. The entry time contained in this record is the object's entry time for a cache of size S.

2Q. The 2Q eviction map maintains two sets of eviction records: one corresponding to the A1 stack, and the other corresponding to the Am stack. While 2Q further partitions the A1 stack into two stacks, A1in and A1out, we model both as a single combined FIFO stack with a single set of eviction records since objects evicted from A1in are placed at the head of A1out. The position in the combined FIFO stack determines whether an object being accessed a second time should be promoted to Am. Each record in the A1 set of eviction records holds the same information as FIFO eviction records and can be used to determine the entry time of an object in the A1 stack. For an object to exist in the Am stack, it must have been accessed at least twice. We can track the number of accesses of each object at varying cache sizes using the same method we used for LFU eviction records. The handling

Algorithm 3: Eviction map for 2Q cache object.

```
    Ref     : object
    Record  : a1_map → Map < cache_size, timestamp >
    Record  : am_map → Map < cache_size, count >
    Record  : Kin, Kout
 1  Eviction Map 2Q:
 2      Function Insert(cache_size):
 3          insert_a1(cache_size ∗ (Kin + Kout))
 4          insert_am(cache_size)
 5      Function FindSmallestExisting():
 6          a1_rec ← a1_map.find_smallest()/(Kin + Kout)
 7          am_rec ← am_map.find_smallest(count ≥ 2)
 8          if !am_rec then
 9              return a1_rec.cache_size
10          return am_rec.cache_size
11      Function GetSortingKey(cache_size):
12          a1in_size ← cache_size ∗ Kin
13          a1out_size ← cache_size ∗ (Kin ∗ Kout)
14          a1in_rec ← a1_map.find_largest(≤ a1in_size)
15          a1out_rec ← a1_map.find_largest(≤ a1out_size)
16          if !a1out_rec then
17              return A1(a1in_rec.timestamp)
18          am_exists ← map.find_any(> a1in_size and
              ≤ a1out_size and object.global_count − count ≥ 2)
19          if !am_exists then
20              return A1(a1out_rec.timestamp)
21          return Am(object.last_access_time)
```

of an access to the eviction map's associated object and the insertion of a new eviction record are done using the same methods previously described for the LFU and FIFO eviction maps. Algorithm 3 contains the pseudocode for the 2Q eviction map's three main operations.

An object will have different a reuse distance depending on whether it is in the A1 or the Am stack. Therefore we calculate two different reuse distances assuming the object is in each stack and select the smaller value (as the cache associated with the larger reuse distance will inherently also contain the object according to the inclusion property). Similar to the FIFO eviction map, we find the smallest A1 stack of size S_{A1} which contains the object by finding the eviction record in the A1 eviction record set with the smallest cache size. The corresponding cache size which contains the object is then $S_{A1}/(Kin+Kout)$. We then search the Am eviction record set for the eviction record with the smallest cache size which has a local count ≥ 2. This eviction record corresponds to the smallest cache of size S_{Am} which contains the object in the Am stack. If no such record exists, the object must exist in the A1 stack and the previously calculated cache size corresponding to the A1 stack is selected as the reuse distance. Otherwise, the reuse distance is $min(S_{A1}/(Kin+Kout), S_{Am})$.

When reconstructing a 2Q stack, we reconstruct the A1 and Am stacks separately. Unlike the previously described policies, an object in a 2Q cache can exist in one of three different stacks: A1in, A1out, or Am. To determine in which stack an object exists for a cache of size S, we use the object's 2Q eviction map to determine its FIFO entry time if it were to exist in the A1in or A1out stack and its local count if it were to exist in the Am stack. As the size of the A1in and A1out stacks are only a ratio of the total cache size, we find the object's entry time in the A1in and A1out stacks for caches of size $S_{A1in} = S * Kin$ and $S_{A1out} = S * (Kin + Kout)$, respectively.[5] If no entry time is found for the object in the A1out stack, it must exist in the A1 stack with the associated A1in entry time. If an entry time is found for the object in the A1out stack, we search the Am eviction records to determine if the object has a local count ≥ 2 for a cache of size $S_{A1in} < S' \leq S_{A1out}$. If such a record exists, it indicates the object has been accessed at least twice while existing in the A1out stack and is therefore in the Am stack. Otherwise, it is in the A1 stack with the associated A1out entry time.

The implementation of an eviction map for the S3-FIFO eviction policy [34] is a simple adaptation of the eviction map for the 2Q eviction policy.[6] We believe a similar technique would work for other multi-stack eviction policies (e.g., ARC [35]) and leave this for future work.

LRFU. The LRFU eviction map is implemented similarly to that of FIFO. Each object in a cache has an associated CRF value (§2.1). Each eviction record in an LRFU eviction map contains the cache size S and the CRF value of the object at S. Such a record identifies the CRF value of the associated object for caches of sizes S', where $S \leq S' < S_{next}$ and S_{next} is the cache size stored in the eviction record with the next largest cache size.

Similar to when using a FIFO eviction map, the reuse distance of an accessed object can be determined using the object's eviction map as the cache size contained in the eviction record with the smallest cache size.

On each access to object O, a new eviction record is added to O's eviction map with S set to 1 and CRF set to $F(0)$. Moreover, the CRF value of each eviction record is updated using the current CRF value and the object's last access time. Each time an object O is evicted, a new eviction record is inserted into O's eviction map. This process is identical to that of a FIFO eviction map.

When reconstructing the stack of a cache of size S, the

[5] Here, we use the combined size of the A1in and A1out stacks when searching for the object's entry time in the A1out stack as the stacks behave as one coherent FIFO stack.

[6] We have also designed eviction maps to support the S3-FIFO eviction policy [34]. It uses three sets of eviction records: one for the "small" stack, one for the "main" stack, and one to track the number of accesses to each object (as with LFU and 2Q eviction records). When reconstructing the S3-FIFO stack, Kosmo reconstructs a separate stack for the "small" and "main" stacks (the "ghost" stack is inherently maintained by the object's existence in Kosmo's global table). When updating the eviction maps of objects, if an object in the "main" stack is selected for eviction though has a local count ≥ 1, it's local count is reduced by 1 and its eviction record is not further updated. We measured similar performance results (including throughput, memory usage, and accuracy) as with that of 2Q in §4.4. Further details on the implementation of Kosmo's eviction map for the S3-FIFO eviction policy and its evaluation are omitted due to space limitations.

objects in the stack are ordered by their CRF values from smallest to largest. To determine the CRF value of an object O in a cache of size S, we find the eviction record with the largest cache size S' in O's eviction map such that $S' \leq S$. The CRF value contained in this record is the object's CRF value at a cache of size S.

LRU. Because the LRU eviction policy adheres to the strict inclusion property, the order of the objects in the simulated caches (of different sizes) will always be the same and can be determined using the objects' last accessed times.[7] By simply storing the reuse distance D of each object as the sole eviction record in its eviction map, Kosmo can reconstruct the stack of a cache of size S by first determining which objects exist in the cache (any object where its reuse distance $D' \leq S$), then ordering the objects that exist by their last access times.

As an object is moved to the front of the LRU stack each time the object is accessed, immediately after, it will exist in the stacks of all caches, regardless of size, until it is again evicted from a cache. Therefore, each time an object is accessed, its associated eviction map updates the object's stored reuse distance to 1 to indicate it now exists at all cache sizes.

Upon each access to an object, to select which objects to evict from the reconstructed cache stacks, an object may be selected for eviction from multiple of these caches. As the object's eviction map has only one eviction record, the object is evicted from the cache with the largest size. In doing so, as LRU adheres to the inclusion property, Kosmo is effectively evicting the object from all caches of smaller sizes as well.

MRU. The implementation of an eviction map for the MRU eviction policy is virtually identical to that of the LRU policy except that the sorting key in the MRU eviction map is the negative value of the object's last access time.

3.6 Variable object sizes

The Kosmo algorithm described thus far generates MRCs assuming the cache is being used for fixed-size objects. However, modern applications use caches to store objects of varying size (e.g., key-value caches). As such, the MRCs generated by these algorithms may not adequately represent the miss ratios experienced by the caches under these workloads. Figure 5 demonstrates the difference in MRCs for the same workload when taking variable-sized objects into account versus not taking them into account. It is evident that these two MRCs differ significantly and variable-sized objects should be accounted for accordingly in MRC generation algorithms.

With fixed-sized objects, Kosmo inserts an eviction record into the eviction map of only one object when a new object is accessed. However, with variable-sized objects, more than one object may need to be evicted. A simple modification

[7] In practice, one would always generate an MRC for the LRU eviction policy using SHARDS and Olken as it is far more efficient than any other known methods. We present a method of generating this MRC using Kosmo simply to demonstrate Kosmo's generality.

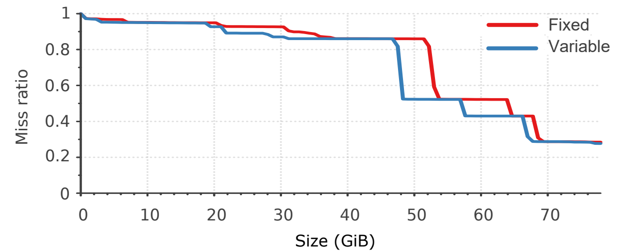

Figure 5: MRCs for the MSR web workload [23] for fixed versus variable-sized objects using the LFU eviction policy.

allows the algorithm to handle variable-sized objects. While a cache's stack is being reconstructed, the cache's used size is calculated by summing the size of all objects which exist in the cache. Objects are then evicted from the top of the stack until the total used size of the cache is less than or equal to the cache's size.

3.7 TTLs

Taking *time-to-live* (TTL) parameters into account can significantly affect the resulting MRC [36]. Minor modifications to the Kosmo algorithm can allow for the support of TTL parameters for objects. When an object is first accessed, a corresponding *expiry time* is calculated based on its TTL. If the TTL is 0, no expiry time is specified. Expiry time describes the time at which the object should be evicted from all caches, regardless of size. To handle this in Kosmo, when iterating through the global table upon each access to reconstruct a cache's stack, each object's expiry time is compared against the current time (i.e., the time of the current access). If the object has not expired and exists in the cache, it is added to the stack; otherwise, it is excluded.

3.8 Simultaneous MRC generation

One key advantage of Kosmo is its ability to generate MRCs for multiple eviction policies simultaneously in a single pass. In the previous descriptions of eviction maps, each object in the global table has one associated eviction map. The type of eviction map used is based on the eviction policy for which an MRC is being generated. A simple extension to the global table to allow each object to have multiple associated eviction maps – one for each eviction policy – allows Kosmo to reconstruct the internal stack of any cache size with any eviction policy for which an eviction map has been defined.

Minor modifications to the previously described Kosmo algorithm (§3.2) must be made to support multi-policy MRC generation. As an MRC must be generated for each eviction policy, the algorithm must maintain a separate histogram per policy. Upon access, an object's reuse distance is calculated for each eviction policy and its corresponding histogram is updated. Moreover, each policy being considered will need its own cache simulations, and each cache's stack is reconstructed in the policy-specific way independently.

4 Evaluation

We evaluated Kosmo using 52 publicly-accessible cache access traces from MSR [23], Twitter [3], and SEC [37, 38]. Table 2 shows a summary of the datasets we used in our evaluation. For the Twitter dataset, we used the recommended traces as specified by Twitter [39] as well as 7 other randomly selected traces in the dataset.[8] Similar to prior studies, we only considered the GET/READ accesses in each trace [20, 40].

For LFU and FIFO, we evaluated Kosmo's performance using all 52 access traces. For 2Q and LRFU, we used all access traces in the MSR and SEC datasets, and 5 randomly selected access traces the Twitter dataset.[9]

We ran both Kosmo and MiniSim with three configurations of SHARDS: one fixed-rate and two fixed-size. The authors of SHARDS noted that for fixed-rate SHARDS, an R value of 0.001, and for fixed-size SHARDS, an S_{max} value of 2,048 produce reasonably accurate MRCs [17]. We selected these same values for our simulations, but also used fixed-size SHARDS with an S_{max} value of 1,024 to examine the memory and throughput benefits as well as the reduction in accuracy. For all fixed-size configurations of SHARDS, we used an initial sampling rate of $R = 0.1$ as we found this produces accurate results.

4.1 MiniSim implementation

We implemented the MiniSim algorithm as described by the original authors of the paper, with the same configuration parameters [18]. The authors only describe MiniSim configured using the fixed-rate SHARDS variant; therefore, we extended MiniSim to also support the fixed-sized SHARDS variant.

Unlike fixed-rate SHARDS, which keeps the sampling rate R constant throughout the access trace, fixed-size SHARDS gradually decreases R to ensure that at any given time there are at most S_{max} distinct objects in the MRC generation algorithm's internal data structures. SHARDS tracks these unique objects in a set S. To extend MiniSim to support fixed-size SHARDS, we initially scale each simulated cache size by the initial sampling rate R. For each access, if the sampling rate R is decreased to R_{new}, we remove all objects no longer in S from all simulated caches. We then rescale the size of each simulated cache by R_{new} using the eviction policy of the cache. A key insight is that when rescaling the size of a simulated cache, we also rescale the cache's access counter (i.e., the number of accesses the cache has observed) and hit counter by the factor R_{new}/R_{old}.

The implementation of MiniSim described in the original paper statically allocates the required memory for each simulated cache before processing an access trace. This is possible as the sampling rate R, and thus the size of each simulated cache, is fixed. In our extension, to support a varying sampling rate, we allocate memory dynamically so as to be able to release memory when R decreases.

Our LFU implementation follows a well-known algorithm optimized for throughput to allow for constant time complexity for each access [41]. Our implementation of 2Q follows that described by the original authors [29]. We used Kin and Kout values of 25% and 50%, respectively, for both our Kosmo and MiniSim simulations. These were the same values used by the original authors. Our implementation of the LRFU eviction policy follows the description in the original paper [27]. We arbitrarily selected a λ value of 0.5 for our experiments though experimented with other values of λ, such as 0.001, and found the results to be similar.

Table 2: Access trace datasets used in our simulations.

Dataset	Access traces	Total accesses
MSR [23]	13	434,212,008
Twitter [3]	24	99,200,180,813
SEC [37, 38]	15	26,482,889,754

4.2 Environment

All experiments were done on Ubuntu 22.04.2 with an AMD Ryzen Threadripper 3990x (64 cores) with 256GB of $DDR4-3200MHZ$ DRAM. The access traces were stored in binary format on a Sabrent Rocket Q 8TB. Both Kosmo and MiniSim use a thread pool with separate threads for each of Kosmo's reconstructed stacks and MiniSim's simulated caches. We tested various thread pool sizes and noticed the best performance for MiniSim when the thread pool's size was equal to the number of cores. Kosmo's performance remained the same after the thread pool's size exceeded the configured granularity.

4.3 Metrics

Three metrics were used in the evaluation of the algorithm: memory usage, throughput, and accuracy.

Memory usage. To measure the memory usage of each algorithm, for each access trace, we ran the algorithm in an isolated process and measured the high water mark [42] after it had processed the entire access trace. This metric has been used in prior work to evaluate the memory usage of MRC generation algorithms [17].

Throughput. To measure the throughput of each algorithm, for each access trace, we divided the total runtime by the number of accesses in the trace. IO time is excluded from the measurement of the total runtime.

Accuracy. To measure the error of both Kosmo and MiniSim, we calculated the mean absolute error (MAE) of each of the generated MRCs using the corresponding exact MRC. As no algorithm exists capable of generating exact MRCs for the LFU, FIFO, 2Q, and LRFU eviction policies, we performed 100 full simulations of caches of varying size (evenly dis-

[8] The randomly selected traces are: `cluster1`, `cluster3`, `cluster8`, `cluster10`, `cluster26`, `cluster50`, and `cluster53`.

[9] The randomly selected traces are: `cluster7`, `cluster22`, `cluster31`, `cluster45`, and `cluster50`.

tributed over the access trace's working set size) for each policy and for each access trace. These 100 points are the same points selected when running MiniSim. To measure the error, we found the MAE by calculating the difference between the exact MRC and the approximate MRCs generated by Kosmo and MiniSim at these points.

4.4 Results

Figures 6-8 show the performance results of Kosmo and MiniSim. For each algorithm, the range of results for the various traces in the datasets is shown.

Figure 6 shows the memory usage results of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found that Kosmo uses an average of 3.6 times less memory, and up to 36 times less in the extreme case. Figure 7 shows the throughput results of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found that Kosmo has an average throughput 1.3 times higher than that of MiniSim. Notably, for the 2Q eviction policy, Kosmo has a lower average throughput than MiniSim (0.54 times that of MiniSim). This is attributed to Kosmo reconstructing two stacks (A1 and Am) on each access.

Figure 8 shows the MAE results of Kosmo and MiniSim for the LFU, FIFO, 2Q, and LRFU eviction policies. We found Kosmo and MiniSim to typically generate MRCs with similar accuracy. Across all simulations, Kosmo and MiniSim had an average MAE within 0.25% of one another. Although Kosmo generates MRCs with lower MAEs, on average, for LFU and LRFU (0.16% and 0.86% lower for LFU and LRFU, respectively), it generates MRCs with higher MAEs, on average, for FIFO and 2Q (0.44% and 1.56% higher for FIFO and 2Q, respectively). This is attributed to the higher rates of violations of the inclusion property for FIFO and 2Q, which we show in §4.5. Further, although the average MAE for the MRCs generated by Kosmo for 2Q is 1.56% higher than those generated by MiniSim, the median is only 0.35% higher. This is attributed to the high MAE of one access trace, `src1` in the MSR dataset [23], which has an unusually high MAE. This access trace violates the inclusion property at a significantly higher rate than other access traces.

To evaluate the CPU usage of Kosmo and MiniSim, we measured the CPU time per access for each access trace. Figure 9 shows that Kosmo's CPU time per access is roughly 1.85 times higher than that of MiniSim for LFU and 2 times higher for FIFO, 2Q, and LRFU. The inconsistency between the lower average CPU time per access of MiniSim than that of Kosmo, and the higher average throughput of Kosmo than that of MiniSim can be attributed to MiniSim's threads idling more frequently than Kosmo's threads.

To evaluate the effects of varying the number of MiniSim's simulated caches on its performance, we also tested MiniSim with 20 and 50 simulated caches for the LFU eviction policy. With 20 simulated caches, MiniSim consumes roughly 1.2 times the memory of Kosmo on average and exhibits roughly

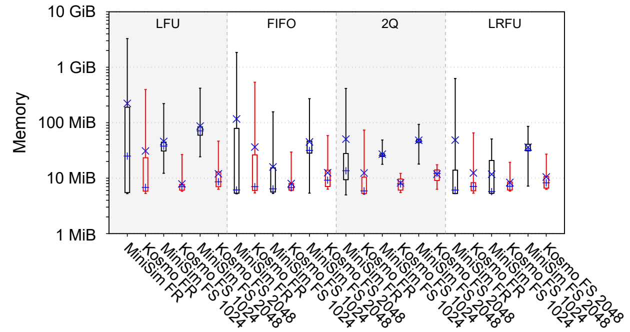

Figure 6: *Memory usage of Kosmo and MiniSim for all eviction policies. Note the logarithmic scale of the y-axis.*

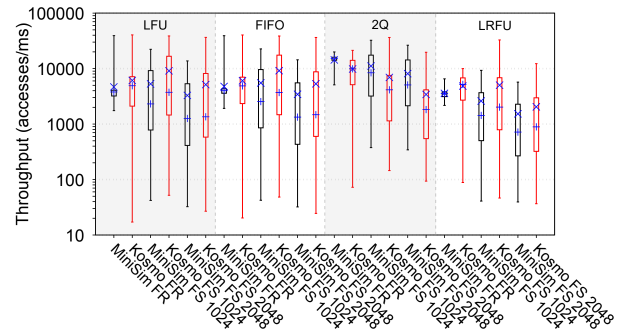

Figure 7: *Throughput of Kosmo and MiniSim for all eviction policies. Note the logarithmic scale of the y-axis.*

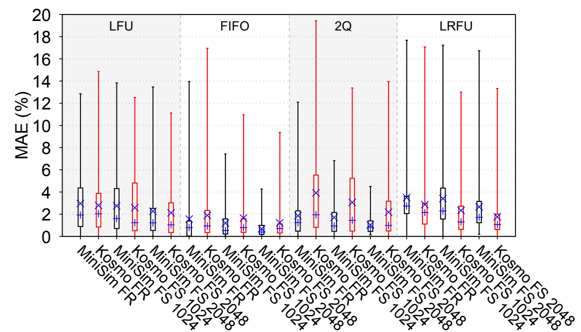

Figure 8: *MAE of Kosmo and MiniSim for all eviction policies.*

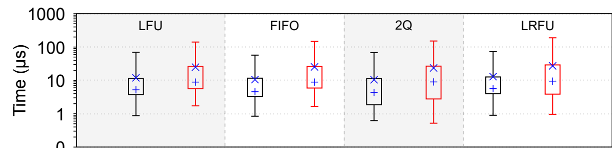

Figure 9: *CPU time per access for the LFU, FIFO, 2Q, and LRFU eviction policies for MiniSim (left, black) and Kosmo (right, red) using fixed-sized SHARDS (S_{max} = 2,048).*

similar throughput, however it has roughly 2 times the MAE of Kosmo. With 50 simulated caches, MiniSim consumes roughly 2.3 times the memory of Kosmo and has 10% lower throughput with roughly identical MAE. Notably, as discussed in §2.3, the C_{max} value of MiniSim must be selected before

knowledge of the access trace, therefore using a low number of simulated caches such as these may result in unobservable points of interest on the MRC.

4.5 Inclusion property violations

Figure 10 shows the percentage of accesses for which a violation of the inclusion property occurs (i.e., accesses which reference an object that does not exist in a cache of size S though exists in a cache of size $S' < S$) for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies across all access traces in the MSR dataset [23]. We found these violations by simulating 100 caches of varying sizes evenly distributed over the access trace's working set size and, for each access, finding the smallest simulated cache in which the object exists, then searching for a larger cache in which it does not.

To examine the severity of these violations, indicated by the difference between the size of the smaller cache wherein an object exists and the larger cache wherein it does not, we repeated these simulations with 50 and 10 simulated caches. This increases the size intervals between the simulated caches and thus, if the number of violations remains high, we can infer the violations occur in large ranges of cache sizes.

For the LFU eviction policy, we found that 1.49% of accesses violated the inclusion property when measured with 100 simulated caches. This reduces by 40.91% and 99.3% to 0.88% and 0.01% when measured with 50 and 10 points, respectively. For the FIFO eviction policy, we found that 20.61% of accesses violated the inclusion property with 100 points, reducing by 18.46% and 77.23% to 16.81% and 4.69% for 50 and 10 points, respectively. For the 2Q eviction policy, we found that 10.49% of accesses violated the inclusion property with 100 points. This reduces by 39.39% and 84.25% to 6.36% and 1.65% for 50 and 10 points, respectively. We found that violations of the inclusion property are rare for the LRFU eviction policy (0.08% of accesses violated the inclusion property with 100 points). The higher rates of violations of FIFO and 2Q can explain the higher MAEs of MRCs generated by Kosmo for these policies, however these errors are typically negligible. Interestingly, we found that MRU violates the inclusion property at a higher rate than the other evaluated eviction policies with an average of 29.12% when simulated with 100 points, while MRU is often considered to not violate the inclusion property [18, 43–45].[10]

5 Related work

Much prior work has focused on improving the performance of in-memory caches [29, 32, 33, 46–55]. Many studies have suggested new eviction policies to improve on observed limitations of policies such as LRU [27, 29, 35, 50, 53, 56–60].

There have been many proposed MRC generation algorithms [14–20, 30, 40, 44, 61–64], however, these are largely

[10] We note that violations of the inclusion property only occur for the MRU eviction policy when considering variable-sized objects.

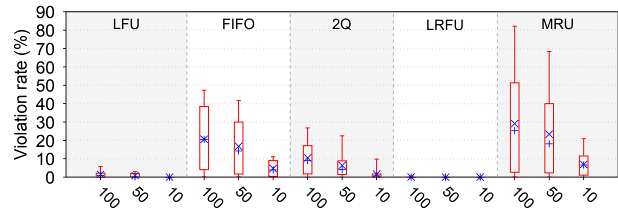

Figure 10: *Ratio of accesses for which a violation of the inclusion property occurs for the LFU, FIFO, 2Q, LRFU, and MRU eviction policies across all workloads in the MSR dataset [23] when simulated with* 100, 50, *and* 10 *points.*

focused on the LRU eviction policy. Beckmann and Sanchez describe a probabilistic method of generating MRCs for other age-based eviction policies [61], such as *protecting distance based policy* (PDP) [53] or *inter-reference gap distribution replacement* (IGDR) [59].

Yu et al. propose an extension to MiniSim, called *DF-Shards*, to modify the number of simulated caches during runtime [44]. We found that the cost of instantiating a new simulated cache significantly reduces the throughput of DF-Shards making it unsuitable for online MRC generation.

MRCs are widely used to improve the performance of caching systems [46, 65–69]. Talus partitions caches to remove identified cliffs in MRCs [46]. Cliffhanger identifies and flattens cliffs noticed in an MRC in real-time, while processing an access trace [32]. mPart uses MRCs to manage the allocation of caches in multi-tenant caching servers [66]. Dynacache also uses MRCs to manage cache allocation; however, the authors also note that modifying the eviction policy in real-time can improve cache performance [68].

6 Concluding remarks

In this paper, we propose Kosmo, a novel method for the simultaneous generation of miss ratio curves (MRCs) for multiple eviction policies. We showed that the current method of generating MRCs for eviction policies that do not adhere to the strict inclusion property have significant memory overhead and are therefore not suitable for online MRC generation. Our experimental results show that Kosmo uses significantly less memory than MiniSim configured with 100 simulated caches while maintaining similar accuracy. Kosmo uses 3.6 times less memory than MiniSim on average, up to 36 times less in the most extreme case. Kosmo has an average throughput 1.3 times that of MiniSim.

In the future, we plan to expand Kosmo's supported eviction policies to more complex policies, such as LHD [50], LIRS [70], or ARC [35]. We also plan to improve on Kosmo's throughput by reducing its computational overhead through the use of more specialized data structures.

Acknowledgements. We thank the reviewers for their constructive comments. We particularly thank our shepherd Carl Waldspurger, whose guidance was instrumental in significantly improving the paper.

A Artifact Appendix

Abstract

The Kosmo artifact provides our implementations of both Kosmo and MiniSim which were used to generate the results presented in this paper. The artifact repository includes three tools: one to calculate an access trace's working set size, one to compute an access trace's accurate MRC for a given eviction policy, and one to generate an approximate MRC (while measuring memory usage, throughput, and accuracy) for both Kosmo and MiniSim. Specific details on each tool's usage can be found in the artifact's README.

Scope

In our evaluation we make the following claims which can be verified by this artifact:

- Kosmo has lower memory overhead than MiniSim by a factor of 3.6 on average, up to a factor of 36.
- Kosmo has a higher throughput than MiniSim by a factor of 1.3 on average.
- Kosmo has a roughly equivalent MAE to MiniSim.

Contents

The artifact compiles to three binaries (further details, including the specific usage of each tool and format of input data is provided in the artifact's README):

1. **wss**: This tool calculates the working set size of a given access trace.
2. **accurate**: This tool runs full simulations to compute the accurate MRC for a given access trace.
3. **mrc**: This tool runs Kosmo or MiniSim (or both) to generate an MRC for a given access trace.

Hosting

The artifact can be found at: 10.5281/zenodo.10569925.

Requirements

The artifact was compiled and tested using `Rust v1.77.0-nightly` and depends on `Gnuplot v5.4` to generate plots.

References

[1] J. Mertz and I. Nunes, "Understanding Application-Level Caching in Web Applications: A Comprehensive Introduction and Survey of State-of-the-Art Approaches," *ACM Computing Surveys*, vol. 50, no. 6, pp. 1–34, Nov. 2017.

[2] A. Wang, J. Zhang, X. Ma, A. Anwar, L. Rupprecht, D. Skourtis, V. Tarasov, F. Yan, and Y. Cheng, "InfiniCache: Exploiting ephemeral serverless functions to build a cost-effective memory cache," in *Proc. Conf. on File and Storage Technologies (FAST'20)*, Feb. 2020, pp. 267–281.

[3] J. Yang, Y. Yue, and K. V. Rashmi, "A Large-Scale Analysis of Hundreds of In-Memory Key-Value Cache Clusters at Twitter," *ACM Transactions on Storage*, vol. 17, no. 3, pp. 1–35, Aug. 2021.

[4] J. Yang, Y. Yue, and R. Vinayak, "Segcache: A memory-efficient and scalable in-memory key-value cache for small objects," in *Proc. Symp. on Networked Systems Design and Implementation (NSDI'21)*, Apr. 2021, pp. 503–518.

[5] Y. Cheng, A. Gupta, and A. R. Butt, "An in-memory object caching framework with adaptive load balancing," in *Proc. of the European Conf. on Computer Systems (EuroSys'15)*, 2015, pp. 1–16.

[6] J. Kwak, E. Hwang, T.-K. Yoo, B. Nam, and Y.-R. Choi, "In-memory caching orchestration for Hadoop," in *Proc. Intl. Symp. on Cluster, Cloud and Grid Computing (CCGrid'16)*, May 2016, pp. 94–97.

[7] Redis Labs, "Redis," https://redis.io.

[8] Memcached, "Memcached," https://memcached.org.

[9] Amazon, "Amazon Web Services," https://aws.amazon.com.

[10] Google, "Google Cloud," https://cloud.google.com.

[11] Microsoft, "Microsoft Azure," https://azure.microsoft.com.

[12] IBM, "IBM Cloud," https://www.ibm.com/cloud.

[13] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch, "Saving cash by using less cache," in *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud'12)*, Jun. 2012.

[14] R. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal*, vol. 9, no. 2, pp. 78–117, 1970.

[15] F. Olken, "Efficient Methods for Calculating the Success Function of Fixed-Space Replacement Policies," Tech. Rep. LBL-12370, May 1981.

[16] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A fast parallel reuse distance analysis algorithm," in *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS'12)*, Aug. 2012, pp. 1284–1294.

[17] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *Proc. Conf. on File and Storage Technologies (FAST'15)*, Feb. 2015, pp. 95–110.

[18] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using Miniature Simulations," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, Jul. 2017, pp. 487–498.

[19] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'16)*, Jun. 2016, pp. 351–364.

[20] J. Wires, S. Ingram, Z. Drudi, N. J. A. Harvey, and A. Warfield, "Characterizing storage workloads with Counter Stacks," in *Proc. Symp. on Operating Systems Design and Implementation (OSDI'14)*, Oct. 2014, pp. 335–349.

[21] B. T. Bennett and V. J. Kruskal, "LRU Stack Processing," *IBM Journal of Research and Development*, vol. 19, no. 4, pp. 353–357, 1975.

[22] D. Carra and G. Neglia, "Efficient miss ratio curve computation for heterogeneous content popularity," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'20)*, Jul. 2020, pp. 741–751.

[23] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," *ACM Transactions on Storage*, vol. 4, no. 3, pp. 1–23, Nov. 2008.

[24] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. Conf. of the IEEE Computer and Communications Societies (INFOCOM'99)*, Mar. 1999, pp. 126–134.

[25] G. Hasslinger, J. Heikkinen, K. Ntougias, F. Hasslinger, and O. Hohlfeld, "Optimum caching versus LRU and LFU: Comparison and combined limited look-ahead strategies," in *Proc. Intl. Symp. on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks (WiOpt'18)*, May 2018, pp. 1–6.

[26] O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's time to revisit LRU vs. FIFO," in *Proc. Workshop on Hot Topics in Storage and File Systems (HotStorage'20)*, Jul. 2020.

[27] S. Min, D. Lee, C. Kim, J. Choi, J. Kim, Y. Cho, and S. Noh, "LRFU: A Spectrum of Policies that Subsumes the Least Recently Used and Least Frequently Used Policies," *IEEE Transactions on Computers*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.

[28] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, M. Tan *et al.*, "Semantic data caching and replacement," in *Proc. Intl. Conf. on Very Large Data Bases (VLDB'96)*, Sep. 1996, pp. 330–341.

[29] T. Johnson, D. Shasha *et al.*, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. Intl. Conf. on Very Large Data Bases (VLDB'94)*, Sep. 1994, pp. 439–450.

[30] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proc. Symp. on Cloud Computing (SOCC'14)*, Nov. 2014, pp. 1–14.

[31] B. Reed and D. D. E. Long, "Analysis of Caching Algorithms for Distributed File Systems," *SIGOPS Operating Systems Review*, vol. 30, no. 3, pp. 12–21, Jul. 1996.

[32] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *Proc. Symp. on Networked Systems Design and Implementation (NSDI'16)*, Mar. 2016, pp. 379–392.

[33] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: A dynamic multi-tenant key-value cache," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, Jul. 2017, pp. 321–334.

[34] J. Yang, Y. Zhang, Z. Qiu, Y. Yue, and R. Vinayak, "FIFO queues are all you need for cache eviction," in *Proc. Symp. on Operating Systems Principles (SOSP'23)*, Oct. 2023, pp. 130–149.

[35] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. Conf. on File and Storage Technologies (FAST'03)*, Mar. 2003, pp. 115–130.

[36] S. Sultan, K. Shakiba, A. Lee, P. Chen, and M. Stumm, "TTLs matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes," Apr. 2024, submitted to Proc. of the European Conf. on Computer Systems (EuroSys'24).

[37] U.S. Securities and Exchange Commission (SEC), "Edgar log file data sets," https://www.sec.gov/about/data/edgar-log-file-data-sets.

[38] J. Ryans, "Using the EDGAR log file data set," 2017. [Online]. Available: https://dx.doi.org/10.2139/ssrn.2913612

[39] Twitter, "Github: twitter/cache-trace," https://github.com/twitter/cache-trace.

[40] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye, "Fast Miss Ratio Curve Modeling for Storage Cache," *ACM Transactions on Storage*, vol. 14, no. 2, pp. 1–34, Apr. 2018.

[41] D. Matani, K. Shah, and A. Mitra, "An O(1) Algorithm for Implementing the LFU Cache Eviction Scheme," *arXiv preprint arXiv:2110.11602*, Oct. 2021.

[42] The kernel development community, "The /proc filesystem," https://www.kernel.org/doc/html/latest/filesystems/proc.html.

[43] X. Gu and C. Ding, "On the theory and potential of LRU-MRU collaborative cache management," in *Proc. Intl. Symp. on Memory Management (ISMM'11)*, Jun. 2011, pp. 43–54.

[44] A. Yu, Y. Tan, C. Xu, Z. Ma, D. Liu, and X. Chen, "DFShards: Effective construction of MRCs online for non-stack algorithms," in *Proc. Intl. Conf. on Computing Frontiers (CF'21)*, May 2021, pp. 63–72.

[45] X. Gu and C. Ding, "A generalized theory of collaborative caching," in *Proc. Intl. Symp. on Memory Management (ISMM'12)*, Jun. 2012, pp. 109–120.

[46] N. Beckmann and D. Sanchez, "Talus: A simple way to remove cliffs in cache performance," in *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA'15)*, Feb. 2015, pp. 64–75.

[47] J. Alghazo, A. Akaaboune, and N. Botros, "SF-LRU cache replacement algorithm," in *Proc. Workshop on Memory Technology, Design and Testing*, Aug. 2004, pp. 19–24.

[48] P. Ranjan Panda, H. Nakamura, N. Dutt, and A. Nicolau, "A data alignment technique for improving cache performance," in *Proc. Intl. Conf. on Computer Design VLSI in Computers and Processors (ICCD'97)*, Oct. 1997, pp. 587–592.

[49] D. Thiebaut, J. Wolf, and H. Stone, "Improving Disk Cache Hit-Ratios Through Cache Partitioning," *IEEE Transactions on Computers*, vol. 41, no. 06, pp. 665–676, Jun. 1992.

[50] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *Proc. Symp. on Networked Systems Design and Implementation (NSDI'18)*, Apr. 2018, pp. 389–403.

[51] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'17)*, Jul. 2017, pp. 499–511.

[52] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in *Proc. USENIX Annual Technical Conf. (USENIX ATC'15)*, Jul. 2015, pp. 57–69.

[53] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum, "Improving cache management policies using dynamic reuse distances," in *Proc. Intl. Symp. on Microarchitecture (MICRO'12)*, Dec. 2012, pp. 389–400.

[54] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE: Dynamic memory management for in-memory data analytic platforms," in *Proc. Intl. Parallel and Distributed Processing Symp. (IPDPS'16)*, May 2016, pp. 383–392.

[55] A. Nasu, K. Yoneo, M. Okita, and F. Ino, "Transparent in-memory cache management in Apache Spark based on post-mortem analysis," in *Proc. Intl. Conf. on Big Data (Big Data'19)*, Dec. 2019, pp. 3388–3396.

[56] M. Bilal and S.-G. Kang, "Time aware least recent used (TLRU) cache management policy in ICN," in *Proc. Intl. Conf. on Advanced Communication Technology (ICACT'14)*, Feb. 2014, pp. 528–532.

[57] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A Highly Efficient Cache Admission Policy," *ACM Transactions on Storage*, vol. 13, no. 4, pp. 1–31, Nov. 2017.

[58] T. B. G. Perez, X. Zhou, and D. Cheng, "Reference-distance eviction and prefetching for cache management in Spark," in *Proc. Conf. on Parallel Processing (ICPP'18)*, 2018, pp. 1–10.

[59] M. Takagi and K. Hiraki, "Inter-reference gap distribution replacement: An improved replacement algorithm for set-associative caches," in *Proc. Intl. Conf. on Supercomputing (ICS'04)*, Jun. 2004, pp. 20–30.

[60] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*, Apr. 1990, pp. 134–142.

[61] N. Beckmann and D. Sanchez, "Modeling cache performance beyond LRU," in *Proc. Intl. Symp. on High Performance Computer Architecture (HPCA'16)*, Mar. 2016, pp. 225–236.

[62] E. Berg and E. Hagersten, "StatCache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. Intl. Symp. on Performance Analysis of Systems and Software (ISPASS'04)*, Mar. 2004, pp. 20–27.

[63] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Proc. Intl. Symp. on Performance Analysis of Systems Software (ISPASS'10)*, Mar. 2010, pp. 55–65.

[64] D. Thiebaut, “On the Fractal Dimension of Computer Programs and its Application to the Prediction of the Cache Miss Ratio,” *IEEE Transactions on Computers*, vol. 38, no. 7, pp. 1012–1026, Jul. 1989.

[65] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar, “Dynamic tracking of page miss ratio curve for memory management,” in *Proc. Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS’04)*, Oct. 2004, pp. 177–188.

[66] D. Byrne, N. Onder, and Z. Wang, “mPart: Miss-ratio curve guided partitioning in key-value stores,” in *Proc. Intl. Symp. on Memory Management (ISMM’18)*, Jun. 2018, pp. 84–95.

[67] M. K. Qureshi and Y. N. Patt, “Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches,” in *Proc. Intl. Symp. on Microarchitecture (MICRO’06)*, Dec. 2006, pp. 423–432.

[68] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, “Dynacache: Dynamic cloud caching,” in *Proc. Workshop on Hot Topics in Cloud Computing (HotCloud’15)*, Jul. 2015.

[69] Z. Liu, H. W. Lee, Y. Xiang, D. Grunwald, and S. Ha, “eMRC: Efficient miss ratio approximation for multi-tier caching,” in *Proc. Conf. on File and Storage Technologies (FAST’21)*, Feb. 2021, pp. 293–306.

[70] S. Jiang and X. Zhang, “LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance,” in *Proc. Intl. Conf. on Measurement and Modeling of Computer Systems (SIGMETRICS’02)*, Jun. 2002, pp. 31–42.