Dynamic Performance Profiling of Cloud Caches

Trausti Saemundsson
Reykjavik University
trausti12@ru.is

Hjortur Bjornsson
University of Iceland
hjb6@hi.is

Gregory Chockler
Royal Holloway,
University of London
Gregory.Chockler@rhul.ac.uk

Ymir Vigfusson
Emory University &
Reykjavik University
ymir@mathcs.emory.edu

Abstract

Large-scale in-memory object caches such as memcached are widely used to accelerate popular web sites and to reduce burden on backend databases. Yet current cache systems give cache operators limited information on what resources are required to optimally accommodate the present workload. This paper focuses on a key question for cache operators: how much total memory should be allocated to the in-memory cache tier to achieve desired performance?

We present our MIMIR system: a lightweight online profiler that hooks into the replacement policy of each cache server and produces graphs of the overall cache hit rate as a function of memory size. The profiler enables cache operators to dynamically project the cost and performance impact from adding or removing memory resources within a distributed in-memory cache, allowing "what-if" questions about cache performance to be answered without laborious offline tuning. Internally, MIMIR uses a novel lock-free algorithm and lookup filters for quickly and dynamically estimating hit rate of LRU caches.

Running MIMIR as a profiler requires minimal changes to the cache server, thanks to a lean API. Our experiments show that MIMIR produces dynamic hit rate curves with over 98% accuracy and $2-5\%$ overhead on request latency and throughput when MIMIR is run in tandem with memcached, suggesting online cache profiling can be a practical tool for improving provisioning of large caches.

Categories and Subject Descriptors C2.4 [*Distributed Systems*]: Distributed applications

General Terms Algorithms, Measurement, Performance

Keywords memcached, profiling, caching, LRU, hit-rate curves, miss-rate curves

SoCC '14, 3-5 Nov. 2014, Seattle, Washington, USA.
ACM 978-1-4503-3252-1.
http://dx.doi.org/10.1145/2670979.2671007

1. Introduction

Distributed in-memory look-aside caches, such as memcached [21] and Redis [1], are pivotal for reducing request latency and database lookup overhead on large websites [19, 36]. In a typical installation, instead of directly querying a database server, web servers will first consult the appropriate memory cache server in a caching tier to check if the response to the database query has already been computed. On a cache hit, the cache server retrieves the query response directly from DRAM and relays the answer back to the web server without involving the database servers. On a cache miss, the query must be sent to a backend database server, typically touching disk or flash, and the response is then written to DRAM at the relevant cache server to accelerate future lookups.

As an instrumental component to the scalability of these large websites, many distributed memory caches handle massive loads: the caching tier of Facebook's social network serves more than 1 billion read requests per second [43]. Challenges related to the scalability of memory caches have been the topic of recent research, including measurement studies [3, 22, 25], optimizations [36, 38, 43] and addressing of issues such as concurrency [19], load imbalance across cache servers [18, 24, 26] and inefficiencies in the underlying software stack [16, 32].

Thus far, however, few inquiries have been made into the provisioning of cache servers [18]. In particular, we lack effective procedures for navigating the trade-off between costs and performance of the caching tier, leaving operators grappling with questions such as:

"*Given the budget and latency service-level-agreements (SLAs) for our website, how many cache servers will we need to operate?*"

For operators hosting sites on elastic cloud services, the problem has a dynamic nature:

"*With the current workload, how many cache servers should we lease from our cloud provider to maintain adequate response time for our customers but still remain cost-effective?*"

Yet understanding this trade-off is crucial for efficient operations. Recently, Zhu *et al.* [50] showed that in typical cloud settings, the ability to scale down both the web server and cache tiers when the incoming request rate subside can save up to 65% of the peak operational cost, compared to just 45% if we only consider scaling down the web server tier. In this paper, we propose MIMIR[1] a new tool for monitoring the cost-performance trade-off for the caching tier. Following Zhu et al., we adopt hit rate – the percentage of cache lookups that return a value and thus bypass a slow database lookup – as our primary performance measure. In terms of cost, the main resource influencing hit rate is the memory capacity of the caching tier, which we will view as the representative dial for controlling cost throughout the paper.

At the core of MIMIR is a profiling algorithm for generating and exposing *hit rate curves (HRC)*, which represent the aggregate cache hit rate as a function of total memory capacity. Hit rate curves allow operators to ask "what-if" questions about their cache tiers using current information, such as to estimate the performance impact of resizing the cache by allocating more servers to the tier, or decommissioning servers. MIMIR hooks into the standard cache replacement API calls of each cache server, and provides the operator with up-to-date HRCs from that server, even for larger memory sizes than currently allocated. The HRCs from different cache servers can then be combined to produce a tier-wide HRC estimate.

Our approach to MIMIR is guided by three main ideas. First, we specifically target the Least-Recently-Used (LRU) replacement policy which allows us to exploit mathematical structure of LRU to generate granular HRCs for the entire cache without tracking each cache size increment separately.

Second, whereas traditional mechanisms attempt to generate HRCs with full fidelity at the cost of performance, we devise a novel bucketing scheme for generating near-exact HRCs that concede between 1-2% of accuracy. Our method has a knob that controls the accuracy versus overhead.

Third, MIMIR piggybacks bookkeeping statistics on the items already stored in memory when predicting HRC for cache sizes smaller than the current allocation. To estimate performance for larger memory sizes, we track dataless keys for recently evicted items, so-called ghosts. As an optimization when key sizes are large relative to the value, we represent the ghost list as an array of lossy counting filters – Bloom-filters that support removal – containing the ghost entries. Together, these techniques allow MIMIR to track HRC with high prediction accuracy and low overhead.

In summary, our paper makes the following contributions.

- We design and implement an architecture for MIMIR that monitors hit rate statistics to help cache operators answer "what-if" questions. At the core of MIMIR is a novel algorithm for dynamically and efficiently estimating the HRC of LRU caches, even in multi-threaded caches.
- Through simulations on a variety of cache traces and benchmarks, we evaluate the accuracy of the MIMIR hit rate profiles, demonstrating that our approach retains high accuracy ($> 98\%$) despite low time and memory complexity. Further, we prove analytically that our algorithm delivers high accuracy on well-behaved workloads.
- As a case study, we modified memcached to use our optimized MIMIR implementation for HRC profiling. Our experimental evaluation shows strong fidelity between the HRC and the true cache hit rate of smaller cache sizes with minor throughput and latency degradation ($< 5\%$).

[1] In Nordic mythology, Mímir was renowned for his knowledge of the future, and was a posthumous consul to the god ruler of Asgard, Odin.

2. MIMIR in a Nutshell

2.1 Architecture

MIMIR is a dynamic profiling framework that interoperates with the replacement policy of a cache service such as memcached. We begin by describing the functionality on a single server; the distributed case is discussed in §2.5. For input, MIMIR provides an API that is called by the replacement policy, as shown in Figure 1. Specifically, a call is made on a HIT, and through the INSERT and DELETE functions when new elements are added or old ones evicted from the cache memory. Each call passes on the identity of the current element that is being manipulated by the cache server. When requested by an operator, MIMIR can generate an up-to-date hit rate curve showing the current estimate of cache hit rate vs. memory size as output. The resulting curves can inform decisions on cache provisioning, either by operators manually or automatically adjusting or partitioning resources [10, 11, 39–41], raise alerts for developers about unexpected cache behavior, and so forth.

2.2 Hit rate curves

At the heart of MIMIR is the HRC estimator. To be precise, we define HRC as the function $H(n)$ for $0 \leq n \leq rN$ denoting the number of hits obtained by a cache replacement algorithm on a cache of size n on a sequence of cache accesses [37]. The function can then be normalized by dividing by the total number of requests in the sequence. If N is the current cache size then, assuming the past predicts the future, $H(n)$ anticipates the hit rate if the cache were resized to a smaller size $n \leq N$, or a larger one $N \leq n \leq rN$ where $r \geq 1$.

Dynamically estimating HRC is fraught with challenges. First, normally cache servers only track the hit rate for the currently allocated memory capacity, for instance: "the 4GB memcached instance currently has hit rate of 73%". They have no mechanism for estimating cache performance for different capacities or configurations. Second, to generate a HRC, the hit rates for different memory sizes must thus be either calculated or simulated. However, calculations depend on the specifics of the cache replacement algorithm, whereas simulations incur significant memory and computation over-

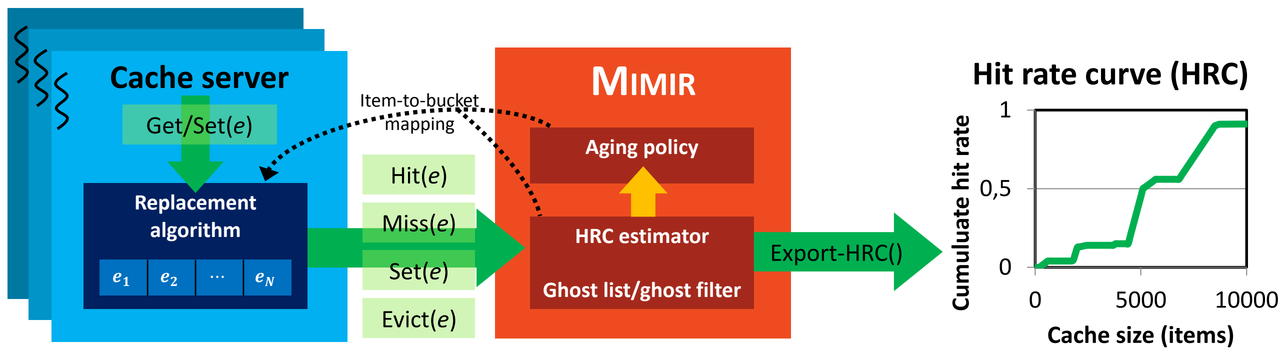


Figure 1: ***The* MIMIR *system.*** *The diagram shows* MIMIR *architecture on a single node. On the left, a multi-threaded cache server such as* memcached *processes* GET(e) *or* SET(e) *cache request for item e. The cache algorithm in the server looks to see if e is among the* $e_1,\ldots,e_N$ *items stored by the cache (a cache hit) or not (a cache miss). In each case,* MIMIR *is given a* HIT(e), MISS(e) *or* SET(e) *upcall. An upcall is also made when the replacement policy evicts an item.* MIMIR *tracks statistics on the elements in the cache, specifically a mapping between buckets and items (see §3). For efficiency, the per-item statistics may be maintained within the cache server as indicated by the dotted line. The keys of elements that have been evicted from the cache may be tracked in a* ghost list *or* ghost filter *to estimate hit rate for larger cache sizes (§2.4). The HRC estimator periodically calls an aging routine to keep the estimate current (§3.3). Finally, as illustrated on the right,* MIMIR *can generate an up-to-date hit rate curve through the export-HRC function when desired by an operator or by a framework using* MIMIR *as input into a resource profiler or cache partitioning framework [11, 39–41].*

heads if done naïvely. Third, since caches are on the critical path for client requests, the estimator must incur low time and space overhead, even if operators wish to assess $H(n)$ for cache sizes n exceeding the current allocation of N.

To make the problem tractable, we focus on estimating HRCs for the LRU cache replacement policy: the default algorithm used in memcached and arguably the most common cache eviction scheme used in practice. Incidentally, LRU has mathematical properties that can be manipulated to efficiently generate HRCs for which we need to define a few additional concepts.

Inclusion property. The principle behind LRU and its descendants (such as CLOCK [13]) is to exploit the locality of memory references. LRU satisfies an *inclusion property* which states that elements in a cache of size N are also contained in a cache of size $N+1$ given the same input sequence and replacement policy [34]. Algorithms that satisfy this condition are called *stack algorithms*, and include LRU, LFU (evict Least-Frequently-Used item) and Belady's OPT (evict the item used farthest in the future) optimal cache replacement policy [6].

Stack distance. One can visualize the contents of an LRU cache over time as a stack, where the item at the bottom of the stack will be replaced on a cache miss. The *stack distance* of an item e is its rank in the cache, counted from the top element. We define elements outside the cache to have an infinite stack distance. The inclusion property induces an abstract total ordering on memory references which stack algorithms use to make replacement decisions. In the case of LRU, elements are ordered by their recency. Therefore, in a linked-list implementation of LRU, the stack distance corresponds to the number of items between the requested element and the head of the list. We will exploit a corollary of the inclusion property and the stack distance definition:

Lemma 1 (from [34])**.** *A request for item e results in a cache hit in an* LRU *cache of size n* if and only if *the stack distance of e is at most the cache size, n.*

2.3 Estimating hit rate curves

Efficiency is a critical design goal for MIMIR: hit rate profiling should have minimal impact on the performance and throughput of cache lookups. After all, minimizing overheads aligns with the performance concerns evidenced by the recent flurry of work on optimizing the memcached service [19, 36, 38].

By Lemma 1, computing the value of hit rate curve $H(n)$ for LRU can be simplified to tracking the number of items with stack distance at most n. Specifically, we can compute $H(n)=\sum_{d=0}^{n}h(d)$ where $h(d)$ is the number of hits received by items of stack distance d. The next question is then how we can efficiently compute and maintain stack distances of cache items.

MATTSON. Mattson et al. [34] proposed a basic approach for computing the stack distance of an item e on a hit: simply traverse the linked LRU list from the head and count the number of elements ahead of e on the list. Unfortunately, the complexity of this basic traversal approach is too high in practice. For instance, Zhao et al. found that on the SPEC CPU2006 benchmark, the $\Theta(N)$ operations for accumulating statistics in MATTSON increased LRU's execution time by 73% [48].

Tree-based schemes. The linear look-up time of MATTSON was recognized as problematic early on. Work towards tree-based alternatives for profiling LRU started with prefix-sum hierarchies proposed by Bennett and Kruskal [7] and culminated in a more efficient approach by embedding the LRU stack in an AVL-tree [2, 46, 48]. Although faster, Zhao

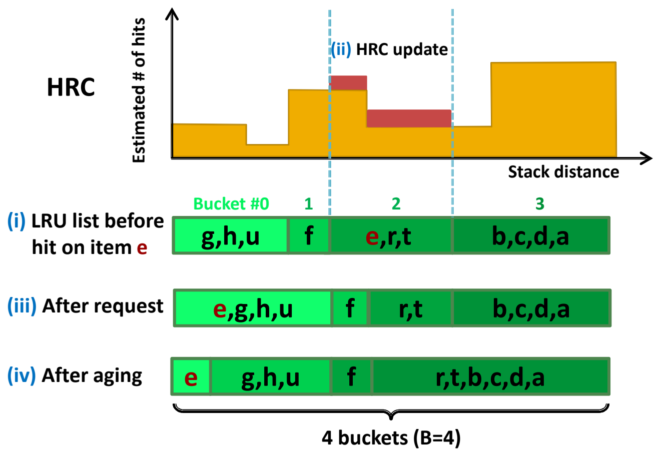


Figure 2: MIMIR's *HRC profiling algorithm.* *The diagram shows updates to the hit rate curve and the bucket lists of the* LRU *stack when element e is hit in the cache.*

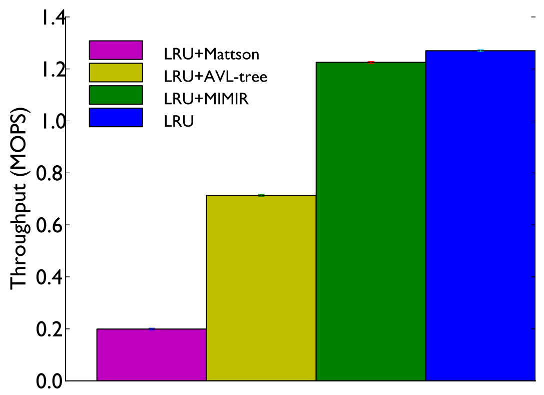


Figure 3: ***Overhead of profiling algorithms.*** *The number of requests processed per second a single-threaded* LRU *C++ cache server with capacity for 5000 items on trace* ***P10****. The error bars show sample standard deviations over 40 runs.*

et al. show that AVL-trees still pose substantial overhead (over 16%) on every cache access [48].

We implemented MATTSON and an optimized AVL-tree LRU profilers on a single-threaded C++ LRU cache server. We compared the performance with and without each profiler on the **P10** trace [35] with a memory capacity for 5000 items. Figure 3 shows that the throughput for MATTSON is only 15.7% of LRU without a profiler, and the AVL-tree comes in at 56.2% of the basic LRU performance. The algorithm for MIMIR we describe below achieves 96.5% of the throughput.

Concurrency. In the context of large memory caches in the cloud, performance issues run deeper than per-item latency overheads. In contrast to the single process environment for which canonical cache algorithms and profiling schemes were developed, modern in-memory cache servers are multi-threaded [19, 21]. Trees and similar data structures facilitate fast reuse distance lookup by maintaining mutable lists that need regular rebalancing, and can therefore suffer from lock contention that limits multi-threaded throughput [9]. Recent work on making concurrent search trees practical is promising, but the implementations still too complicated for deployment within an LRU profiler [9]. The performance results in Figure 3 coupled with the concurrency concerns prompted us to consider alternative approaches to stack distance profiling.

MIMIR's profiling algorithm. The above approaches are based on the premise that all stack distances must be calculated accurately. In practice, however, decisions based on HRCs are made at coarser granularity, such as for cache partitioning [40, 41] or server provisioning. If we concede to producing approximate HRCs, how accurate can they be if our methods need to be fast and concurrent?

We devised a novel estimation algorithm for HRC that tracks approximate LRU stack distances. We imagine that the LRU stack has been divided into B variable-sized *buckets*, where B is a parameter, and that every element tracks the identity of the bucket to which it belongs. The elements can be in any order within a bucket.

Our method exploits a characteristic that enables us to hone in on the true LRU stack distance of an item. Suppose the buckets are labeled $\mathbf{B}_0, \mathbf{B}_1, \ldots, \mathbf{B}_{B-1}$ where $\mathbf{B}_0$ is closest to the LRU head, and define $n_0, n_1, \ldots, n_{B-1}$ as the corresponding sizes of the buckets such that $\sum_{i=0}^{B-1} n_i = N$, where N is the cache size. The invariant we maintain is that an element e_i in bucket $\mathbf{B}_i$ will have lower stack distance than element e_j in bucket $\mathbf{B}_j$ if and only if $i < j$. Moreover, the true LRU stack distance of elements in bucket $\mathbf{B}_j$ will be in the range between L and $L + n_j - 1$ where $L = \sum_{i=0}^{j-1} n_i$. When bucket sizes are reasonably balanced, we can estimate the histogram of stack distances as a probability distribution over these intervals, and accumulate the probability distributions over requests to estimate the HRC.

A high-level illustration of our algorithm with $B = 4$ buckets is shown in Figure 2. Suppose a cache request is made for element e for the LRU stack shown on label **(i)** on the figure. From e's identifier, we see it belongs to bucket #2, which also contains two other elements. The invariant implies that e's stack distance must lie in the range $[4, 6]$, since the two buckets on the left contain four elements and e is one of the three elements in bucket #2.

Since we have no further information about where e lies in the range, we assume it to be uniformly distributed within the interval. We therefore update the estimated number of hits at stack distances 4, 5 and 6 by a value of $\frac{1}{3}$ each (label **(ii)**). Element e is now moved to the front of the list and tagged as belonging to bucket #0 (label **(iii)**). To ensure load is balanced across the buckets, the bucket contents are shifted down the list by an *aging* procedure when the front bucket holds more than its fair share ($\lceil \frac{N}{B} \rceil = \lceil \frac{11}{4} \rceil = 3$) of items (label **(iv)**). We detail properties of the algorithm and two aging procedures (ROUNDER and STACKER) in §3.

2.4 Estimating HRC for larger cache sizes

The above techniques are designed to calculate $H(n)$ for n at most the cache size N since they rely on the ability to detect cache hits. The next question is whether one can estimate $H(n)$ for $n > N$, allowing one to determine whether cache needs to grow in size.

Ghost lists. To this end, one approach is that upon an eviction, we keep track of the identifier of the evicted item and discard the value. These dataless items are called *ghosts* [17]. By keeping a fixed number of such recently evicted items on a *ghost list*, one can record accesses to items whose data *would* have been included in the cache if more memory had been available [37].

Normally, a ghost consumes negligible amount of memory relative to the data stored by original items. Thus the memory overhead of a ghost list can be viewed as "tax" on the elements stored in the cache. When an item is evicted from the primary LRU list, it is added to a ghost LRU stack. This ghost stack can now be used to estimate stack distances above N according to Lemma 1. To keep the ghost list from growing unbounded, the last ghost is also popped off the stack and discarded. By treating the ghost LRU stack as an extension of the primary one, the profiling algorithms described above will work without change. In the MIMIR profiling algorithm, for instance, the LRU list is partitioned into buckets that are occupied by regular items and ghosts alike. To summarize, although hits on ghost entries are cache misses, since no data can be returned to the user, ghosts allow us to estimate large stack distances and therefore expand the domain of the HRC.

The management of ghost lists incurs performance overhead. While there is no penalty on cache hits, cache misses now require a table lookup, and extra bookkeeping akin to the one performed on a regular cache hit. When miss rates are significant, this overhead can degrade the cache server throughput. However, ghosts have less effect on per-request latency incurred by clients since cache misses require retrieval from secondary storage and are thus already slow.

Ghost filters. In workloads where values are short relative to item keys, the memory tax for maintaining ghost items can be significant. Such situations happen in practice: a recent survey by Facebook researchers showed that tiny values (11 bytes or less) are common in their workloads, comprising more than 40% of the values sampled from the largest memcached pool (ETC) in the study [3].

We can reduce the memory overhead by arranging ghost lists into a fixed number of counting Bloom filters [20], each representing a single MIMIR bucket. The list of filters is a queue maintained in a circular array. The key of an item undergoing eviction is added to the head filter. When the head filter is full, the filters are rotated down the list and the tail filter is discarded and a new head filter is created. On a miss, we query each of the filters for the key of the item. If the key is found, it is removed from the filter and hit rate statistics are updated in the same fashion as in the MIMIR algorithm above.

2.5 Combining estimates

Having described how the HRCs are generated on a single cache server, we can now address the distributed case. If the caching capacity allocated to an application is distributed across k servers in the caching tier, the HRCs produced by MIMIR on individual servers can be joined to produce a single tier-wide HRC as follows.

Every server i maintains an array of hit statistics H_i where each entry $H_i(j)$ for $j \geq 0$ holds the number of hits obtained by a cache replacement algorithm on a cache of size j along with the total number of hits N_i. To produce a combined HRC, H, observe that by assuming the items are spread between the caches uniformly at random [30], each $H_i(j)$ represents server i's portion of the total number of hits obtained by the combined cache of size $k(j+1)$. Hence, for each $j \geq 0$, $H(k(j+1)) = \sum_{i=0}^{k-1} H_i(j)$. For simplicity, we fill the "gaps" between each pair of combined cache sizes kj and $k(j+1)$, by assigning the same number $H(k(j+1))$ of hits to each cache size within this range resulting in the combined HRC being a step function. Note that other approaches, such as piecewise linear approximation, could also be applied. Finally, we normalize the combined HRC as $\bar{H}$, obtained by dividing each entry in H by $\sum_{i=0}^{k-1} N_i$.

3. The MIMIR Algorithm

At the center of MIMIR is the HRC profiling algorithm we outlined in §2.3. Our algorithm is designed to generate estimates for LRU stack distances while incurring minimal overhead, even when deployed in multi-threaded settings. While we focus on the in-memory caching tier as our target setting, our algorithm can also be applied in other memory management scenarios, such as facilitating memory partitioning between virtual machines [23, 29, 33, 47] or applications [31, 49]. We will now discuss specifics and optimizations for the HRC estimator, and formally analyze their running time and accuracy.

3.1 Preliminaries

Recall that our method relies on maintaining a dynamic partition of all currently cached elements into a fixed number B of *buckets*. The buckets are logically organized into a circular list with the most and least active buckets occupying the *head* and *tail* of the list, respectively. Whenever a cache hit occurs, the element causing the hit is moved to the head of the list. The total number of elements in the buckets in front of the one being hit is then used to estimate the stack distance of the hit, as we explain below.

To keep the elements in our buckets consistent with their stack distances, the elements are *aged* by shifting them one bucket down in the bucket list. Because we do not differentiate between the stack distances of the elements

Figure 4: *Pseudocode for (**B**)* MIMIR*'s HRC estimator algorithm and the (**C**)* STACKER *and (**D**)* ROUNDER *aging policies.*

```
(A) Initialization
Record Element:
    $rank$, initially undefined ;Bucket in which item is associated:
$buckets[B] \leftarrow [0,\dots,0]$ ;Cyclic array of per bucket item counts
$tail \leftarrow 0$ ;Index of the tail bucket
$AvgStackDist \leftarrow 0$ ;Estimated average stack distance touched since latest aging round (used by STACKER)
$DELTA[N] \leftarrow [0,\dots,0]$ ;Cumulative hit count statistics
$HRC[N] \leftarrow [0,\dots,0]$ ;HRC estimate
```

```
(B) MIMIR Estimator Algorithm
procedure HIT($e$)
    if $e.rank < tail$ then
        $e.rank \leftarrow tail$ ;adjust last bucket for ROUNDER
    $(start, end) \leftarrow get\text{-}stack\text{-}dist(e)$
    $update\text{-}HRC(start, end)$
    $i \leftarrow e.rank \bmod B$
    $buckets[i] \leftarrow buckets[i] - 1$
    $head \leftarrow (tail + B - 1) \bmod B$
    if $buckets[head] = N/B$ then
        $age()$
    $e.rank \leftarrow tail + B - 1$
    $buckets[head] \leftarrow buckets[(tail + B - 1) \bmod B] + 1$

procedure INSERT($e$)
    $head \leftarrow (tail + B - 1) \bmod B$
    if $buckets[head] = N/B$ then
        $age()$
    $e.rank \leftarrow tail + B - 1$
    $buckets[head] \leftarrow buckets[head] + 1$

procedure DELETE($e$)
    if $e.rank < tail$ then
        $e.rank \leftarrow tail$ ;adjust last bucket for ROUNDER
    $i \leftarrow e.rank \bmod B$
    $buckets[i] \leftarrow buckets[i] - 1$

procedure $get\text{-}stack\text{-}dist(e)$
    $(start, end) \leftarrow (0, 0)$
    $(head, tail) \leftarrow (tail + B - 1,\ tail)$
    for $i \leftarrow head \rightarrow tail$ do
        if $i = e.rank$ then
            break
        $start \leftarrow start + buckets[i]$
    $end \leftarrow start + buckets[e.rank \bmod B]$
    return $(start, end)$

procedure $update\text{-}HRC(start, end)$
    $delta \leftarrow 1/(end - start)$
    $DELTA[start] \leftarrow DELTA[start] + delta$
    $DELTA[end] \leftarrow DELTA[end] - delta$

procedure $export\text{-}HRC()$
    $HRC[0] \leftarrow DELTA[0]$
    for $i = 1 \rightarrow N - 1$ do
        $HRC[i] \leftarrow HRC[i-1] + DELTA[i]$
```

```
(C) STACKER aging policy
procedure $age()$
    $b \leftarrow get\text{-}bucket(AvgStackDist)$
    for all elements $e$ in the cache do
        $i \leftarrow (e.rank) \bmod B$
        if $i \le b$ then
            $buckets[i] \leftarrow buckets[i] - 1$
            $e.rank \leftarrow e.rank - 1$
            $buckets[i-1] \leftarrow buckets[i-1] + 1$

procedure $get\text{-}bucket(d)$
    $len \leftarrow 0$
    for $i \leftarrow B - 1 \rightarrow 0$ do
        $len \leftarrow len + buckets[i]$
        if $d \le len$ then
            break
    return $i$
```

```
(D) ROUNDER Aging Policy
procedure $age()$
    $buckets[(tail + 1) \bmod B] \leftarrow buckets[(tail + 1) \bmod B] + buckets[tail \bmod B]$
    $tail \leftarrow tail + 1$
    $buckets[(tail + B - 1) \bmod B] \leftarrow 0$
```

mapped to the same bucket, aging is only necessary when the head bucket fills (with N/B elements). This allows us to both reduce the frequency of aging to at most once per N/B requests, and amortize its overhead.

In our implementation, we maintain a correspondence between the elements and the buckets in the following three state variables (see Algorithm **A**):

(1) An $e.rank$ tag associated with each cached element e is used to establish the identity of the e's current bucket.

(2) A circular array *buckets* tracks the number of elements currently mapped to each bucket.

(3) The *tail* variable holds the current index of the lowest bucket in the *buckets* array.

Initially, $tail = 0$, $buckets[i] = 0$ for all $0 \le i < B$, and $e.rank$ is undefined.

The MIMIR implementation divides the logic of LRU stack processing into an *HRC estimator*, which intercepts and processes the calls made to the MIMIR's framework, and an *aging policy*, which balances the load between buckets and manages the *tail* index. We assume that the implementation of the replacement algorithm (see Figure 1) ensures that each call to HIT (or MISS) can only be made for the elements that have been previously inserted into the cache (through the INSERT call), and are still active at the time the HIT (or MISS) is invoked.

To simplify presentation, we defer the details of the ghost list management to the full paper (see Section 2.4 for the informal discussion), and focus solely on estimating HRC for the currently occupied caching space. Consequently, the HRC estimator code in Algorithm **B** only includes the details of processing the HIT, INSERT, and DELETE calls, and omits those of the MISS call.

For output, the hit statistics are collected into a *DELTA* array whose entries $DELTA[i]$ hold the differences between the consecutive elements of the hit histogram. Using the *DELTA*, we can derive an HRC density function via the *export-HRC* routine, which assigns $\sum_{i=0}^{k} DELTA[i]$ to each $HRC(k)$ for all $0 \le k \le N$ where N is the size of the cache. The hit rate curve is generated through cumulative summation of the ensuing *export-HRC* density function.

Below, we discuss the MIMIR algorithm with two aging policies, STACKER and ROUNDER, representing different trade-offs between estimation accuracy and efficiency.

3.2 HRC estimator

When the cache replacement algorithm receives a hit on element e, the MIMIR HRC estimator's HIT routine is invoked with e as a parameter, and proceeds as follows (see Algorithm **B**): First, the *get-stack-dist* routine determines the lower (*start*) and upper (*end*) stack distance boundaries of the bucket in which e is currently contained. This is done by summing up the entries in the *buckets* array starting from the topmost entry until the bucket whose index is equal to $e.rank$ is reached (see the *get-stack-dist* routine). Our invariant implies that the relative stack distance order of items within the bucket is unknown. Therefore, we cannot count the hit by simply adding 1 to the array at the precise stack distance of e. We will instead assume a hit is equally likely to occur at any stack distance between *start* and *end*. Accordingly, we add $\frac{1}{end-start}$ to each distance in the interval $[start, end]$.

Next, *e*'s mapping is adjusted by moving it to the topmost bucket $(tail+B-1)$ mod B, and then adjusting the affected *buckets* array entries to reflect the update. Before making the move, the HIT handler checks if the number of elements in the head bucket would consequently exceed N/B. If so, we balance bucket sizes by invoking the aging policy, which will empty the head bucket and possibly shift the elements in other buckets to an adjacent bucket further back in the list. The details of the aging policies are detailed below.

The HRC estimator is notified of an element *e* newly inserted into the cache via the INSERT routine which adds *e* to the topmost bucket, possibly aging the elements if the topmost bucket is full. Finally, deleting an element *e* from the cache triggers the DELETE(e) handler, which removes *e* from the bucket in which it is currently located.

3.3 Aging Policies

In this section, we describe the details of the two aging policies, called STACKER and ROUNDER respectively, that can be used in conjunction with the MIMIR's HRC estimator.

STACKER. The pseudocode for STACKER aging policy appears in Algorithm **C**. The idea behind the method is to track the average stack distance that has been accessed, and approximating the LRU aging by only shifting the items whose stack distance is less than the average. To this end, the algorithm maintains a variable *AvgStackDist* holding the weighted running average of the stack distances that have been accessed so far.

The aging then proceeds as follows. First, the average stack distance is converted to the bucket number in the *get-bucket* routine by finding the first (from the top) bucket *b* in the bucket list such that the total number of the elements in the buckets preceding *b* is at least as high as *AvgStackDist*. The elements in the cache are then traversed in a top-down fashion, decrementing their *e.rank* value, and adjusting the affected bucket counts for each element *e* whose *e.rank* is at most *b*.

Note that the STACKER's *age* routine never updates the value of *tail*, thus leaving the tail bucket coinciding with *buckets*$[0]$ for the entire duration of the algorithm's run.

ROUNDER. Observe that the running time of the STACKER aging policy is linear in the number of the cached items, which can be quite expensive for large cache sizes. To address this shortcoming, the ROUNDER aging policy replaces the downward shift of the cached elements with advancing the current tail bucket index in the circular *buckets* array, thus effectively *shifting the frame of reference of bucket identifiers*. This shift is implemented in three steps (see Algorithm **D**). First, the current tail bucket count is folded into the one immediately after. Then, the *tail* is incremented thus turning the former tail bucket into the head one. Finally, the new head bucket is emptied by setting its entry in the *buckets* array to 0.

Following this transformation, all elements with rank tag value of $tail+B-1$ will be correctly mapped to the new head bucket. However, the new tail bucket will end up populated by the elements from the old tail. To map these elements to correct buckets on either HIT or DELETE we adjust their rank tags to point to the current *tail*.

3.4 Maximizing concurrency

The simple data structure and aging mechanism used by ROUNDER makes our HRC estimator highly amenable to a concurrent implementation, boosting performance on modern multi-processor architectures. We implemented a *non-blocking* version of ROUNDER, where no thread, even a slow one, can prevent other threads from making progress.

First, we update bucket counters atomically using standard *compare-and-swap (CAS)*-based technique whereby the update is retried until it is verified that the new counter value is derived from the most recently read one.

Second, we guarantee that at most one of the threads executing within the *age* routine for a given value of *tail* can succeed to update the head and tail buckets, and advance the tail, as follows. The most significant bit of each bucket counter is reserved to serve as a *mark* indicating whether the bucket was successfully set as a tail bucket by a recent update. The threads executing within the *age* routine then attempt to update both the penultimate bucket counter and its mark in a single atomic step using CAS until one of the threads succeeds. This thread will then proceed to both update the head and advance the tail whereas all other threads will discover the mark set and return. This mechanism guarantees that all threads invoking *age* for a given value of *tail* will eventually return, and all the aging updates will be executed exactly once as needed.

3.5 Complexity analysis

Let $N > 0$ be the maximum cache capacity. As we discussed earlier, the MATTSON and AVL-trees profiling algorithms incur $\Theta(N)$ and $\Theta(\log N)$ overhead on every cache hit, respectively. In contrast, the complexity of our HRC estimator is bounded by the number of buckets *B*, effectively a constant.

Specifically, the running time of the HIT handler is dominated by the *get-stack-dist* routine, which may take at most $O(B)$ steps in the worst case. In addition, the STACKER aging policy (see the *age* routine in Algorithm **C**) takes $O(B+N)$ time to complete, which includes $O(B)$ steps of the *get-bucket* routine, and additional $O(N)$ iterations of the aging loop. The complexity of the ROUNDER aging policy is $O(1)$.

Thus, the maximum hit processing time of MIMIR is either $O(N)$ or $O(B)$ for the STACKER and ROUNDER aging policies respectively. Furthermore, since the *age* routine is executed no more than once per N/B requests, and $N \geq B$, the MIMIR's amortized time complexity per hit is just $O(B)$ for both STACKER and ROUNDER. If ROUNDER is used, the amortized complexity can be further reduced to $O(1)$ if $N \geq B^2$, and tends to 0 if $N \gg B^2$.

Since, as we show in §3.6 and §4, the values of B as small as 8 or 16 are sufficient to obtain high quality HRC estimates for caches of any size, the hit processing times of our algorithms are essentially bounded by a small constant.

Further optimizations are possible. We can bound STACKER's processing time on hits by limiting the maximum number of elements that can be processed at each invocation of the *age* routine, or by delegating the aging loop execution to a background thread. We can also speed up the stack distance computation of *get-stack-dist* by using a tree of partial sums of the *buckets* array entries to quickly compute the lengths of the relevant stack segments resulting in the $O(\log B)$ execution time [7].

3.6 Average error analysis

Our methods trade off prediction accuracy for performance by assessing statistics at a granularity of B buckets. Having discussed the asymptotic complexity of the algorithm, we now turn to analyzing the prediction error. We use the mean average error MAE of a distribution $h : [1,N] \to [0,1]$ relative to an optimal distribution h^*:

$$\mathrm{MAE}(h,h^*) = \frac{1}{N}\sum_{x=1}^{N} |h(x) - h^*(x)| . \qquad (1)$$

Note that our results are robust against other ℓ_p-norms. We can derive an upper bound on the mean average error of the estimated HRC for STACKER given a trace of requests.

Theorem 1. *For an* LRU *cache of size N during a trace of R requests, With B buckets, our algorithm has a mean average prediction error (MAE) bounded by the largest bucket size during the trace, divided by N/2. Consequently, if no bucket grows larger than $\alpha N/B$ for $\alpha \geq 1$ during the trace, then the MAE for our approach is at most $\frac{2\alpha}{B}$.*

Proof. We consider R cache requests to have true reuse distance $r_1, r_2, \ldots, r_R$. It suffices to consider only requests that result in LRU hits, so $r_i \leq N$ for all i. Define $\delta_t(x) = 1$ if $x = r_t$ and $\delta_t(x) = 0$ otherwise. Then the optimal LRU hit rate curve HRC* satisfies:

$$\mathrm{HRC}^*(x) = \frac{1}{R}\sum_{t=1}^{R}\sum_{z=0}^{x} \delta_t(z)$$

In STACKER, there are B buckets with variable boundaries over time. For request t with true reuse distance r_t, we estimate the reuse distance over an interval $[a_t, b_t]$ that includes r_t. Furthermore, we assign uniform probability to all possible distances within that interval. Define $c_t(x) = \frac{1}{b_t - a_t}$ when $x \in [a_t, b_t)$ and $c_t(x) = 0$ otherwise. Then the hit rate curve for our algorithm satisfies:

$$\mathrm{HRC}(x) = \frac{1}{R}\sum_{t=1}^{R}\sum_{z=0}^{x} c_t(z)$$

Using the triangle inequality, we obtain the following upper bound on the mean average error for the two HRCs.

$$\begin{aligned}
\mathrm{MAE}(\mathrm{HRC},\mathrm{HRC}^*) &= \frac{1}{N}\sum_{x=0}^{N-1} |\mathrm{HRC}(x) - \mathrm{HRC}^*(x)| \\
&= \frac{1}{NR}\sum_{x=0}^{N-1}\left|\sum_{t=1}^{R}\sum_{z=0}^{x} \delta_t(z) - c_t(z)\right| \\
&\leq \frac{1}{NR}\sum_{t=1}^{R}\sum_{x=0}^{N-1}\sum_{z=0}^{x} |\delta_t(z) - c_t(z)| \\
&= \frac{1}{NR}\sum_{t=1}^{R}\sum_{x=a_t}^{b_t}\sum_{z=0}^{x} |\delta_t(z) - c_t(z)| \\
&\leq \frac{1}{NR}\sum_{t=1}^{R}\sum_{x=a_t}^{b_t}\sum_{z=0}^{x} (|\delta_t(z)| + |c_t(z)|) \\
&\leq \frac{1}{NR}\sum_{t=1}^{R}\sum_{x=a_t}^{b_t} (1+1) = \frac{2}{NR}\sum_{t=1}^{R} b_t - a_t .
\end{aligned}$$

□

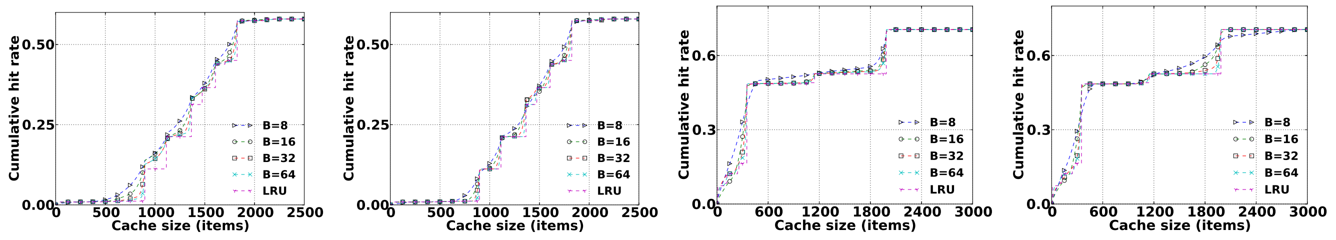
The average bucket size for hits $\frac{2}{R}\sum_{t=1}^{R}(b_t - a_t)$ can be calculated during the execution of the algorithm. We computed the average bucket size for hits in our simulations and found that it provides a loose but useful bound on the MAE without needing to compute the optimal HRC*. The average can also be bounded above by the largest bucket that receives hits during the trace, $\frac{2}{N}\sup_{t=1,\ldots,R}(b_t - a_t)$, but our experiments indicate that the least-significant bucket which tracks the largest reuse distances can consume a significant portion of the cache.

In summary, the analytic result enables operators to dynamically track the MAE of the HRC estimate without computing the optimal hit rate curve. The algorithm could thus be extended to adaptively add or remove buckets depending on changes in the MAE so as to maintain accuracy.

3.7 Extensions

Items of different sizes. The HRC estimator algorithm implicitly assumes that cached elements are all of the same size. In other words, the contribution of each hit to the HRC is assumed to be independent of the space occupied by the element being accessed. The approach can be extended easily to support different element sizes by expressing the size of each element in terms of *pages* of fixed size (similar to virtual memory pages), and then treat each hit of an element of size p pages as p simultaneously occurring individual hits with one hit per page. When updating the hit contribution in line 35 of Algorithm **B**, we would instead compute the value as $p/(end - start)$. In addition, the code computing the value of *end* in *get-stack-dist* must be adjusted to address the case of elements spanning multiple buckets.

Adaptive profiling. To adapt the HRC estimation to dynamically changing cache access patterns, the hit statistics stored in the *DELTA* array are periodically aged using an



(a) ROUNDER *on trace file* ***glimpse*** **(b)** STACKER *on trace file* ***glimpse*** **(c)** ROUNDER *on trace file* ***postgres*** **(d)** STACKER *on trace file* ***postgres***

Figure 5: ***Accuracy on microbenchmarks.*** *Hit rate curves of* ROUNDER *(top row) and* STACKER *(bottom row) with varying bucket sizes (B) running over* LRU *replacement on two representative trace files. The true* LRU *hit rate curve is also shown.*

a priori chosen aging factor parameter γ, $0 < \gamma < 1$. More specifically, we collect statistics in epochs of equal duration. At the end of each epoch, the values stored in the *DELTA* array are folded into the HRC using the *export-HRC* routine, and the *DELTA* array entries are multiplied by γ. The value of γ reflects the trade-off between the speed of the HRC adaptation, and its resistance to short-lived fluctuations in the workload. In our experiments, $\gamma = 0.1$ was a good match for the workload types we analyzed.

4. Evaluation

Our evaluation on the MIMIR HRC algorithm focuses on quantifying the following questions.

- **Accuracy.** What is the degree of fidelity between the optimal HRC and the estimates produced by the algorithm? How different is the precision between ROUNDER and STACKER? How is it affected by algorithm parameters?
- **Performance.** What is the performance overhead of the estimation algorithm? To what extent does the gathering of statistics degrade the cache throughput?

Methodology. We begin by assessing the accuracy of our methods through simulations. We implemented a trace-driven cache simulator in Python and C++ and measured the algorithms on a variety of standard cache traces and benchmarks. We then experiment with MIMIR running with memcached, and measure the impact on memcached's throughput and latency. We also compare the accuracy of our method compared with the optimal MATTSON algorithm, and look at the prediction accuracy. Many results are similar, so we present only representative traces and benchmarks.

4.1 Simulations

We ran simulations on a variety of traces from the cache literature to measure the estimation quality of our methods. As the primary metric for assessing quality, we use the mean average error (MAE, eq. 1) – the proximity between h: the HRC generated by an algorithm, and h^*: the optimal LRU hit rate curve generated by MATTSON. The MAE between two distributions ranges from 0% for identical curves to at most 100% for complete dissimilarity. Accuracy is defined as $1 - \text{MAE}(h, h^*)$.

Workloads. We use traces and benchmarks that are commonly used by the cache replacement algorithm community [27, 28, 35], with the parameters outlined in Table 1. The traces present challenging workloads whose hit rate curves are hard to approximate. The difficulty stems from abundance of sequential and looping references in the buffer cache that are characteristic of file-system accesses [31].

The traces **2-pools**,**glimpse**, **cpp**, **cs**, **ps** and **sprite**, were respectively collected from a synthetic multi-user database, the `glimpse` text information retrieval utility, the GNU C compiler pre-processor, the `cs` program examination tool, join queries over four relations in the `postgres` relational database, and requests to file server in the Sprite network file system [27, 28]. The traces **multi1**, **multi2** and **multi3** are obtained by executing multiple workloads concurrently [27]. For these first nine traces, we use identical cache set-up as the authors of the LIRS algorithm [27].

We also used workloads captured from IBM SAN controllers at customer premises [35]. The workloads **P1**-**P13** were collected by disk operations on different workstations over several months. **WebSearch1** consists of disk read accesses by a large search engine in response to web search requests over an hour, and **Financial1** and **Financial2** are extracted from a database at a large financial institution [35].

Profiling accuracy. Figure 6 summarizes the quality of the estimates for MIMIR on the workloads with both STACKER and ROUNDER aging policies as we vary the numbers of buckets B. We observe that the average accuracy exceeds 96% for both methods on all traces, and for STACKER with 128 buckets the accuracy is 99.8% on average over all the traces. The bucket parameter trades off overhead for more granular reuse distance estimates, and the improvement in accuracy is evident as we increase the number of buckets B. The gradual improvement of the approximation can be seen in Figure 5 on some canonical traces. The extra overhead of STACKER over ROUNDER translates consistently to better HRC predictions.

Unlike the more expensive STACKER aging policy, ROUNDER had the poorest performance on the **WebSearch1**

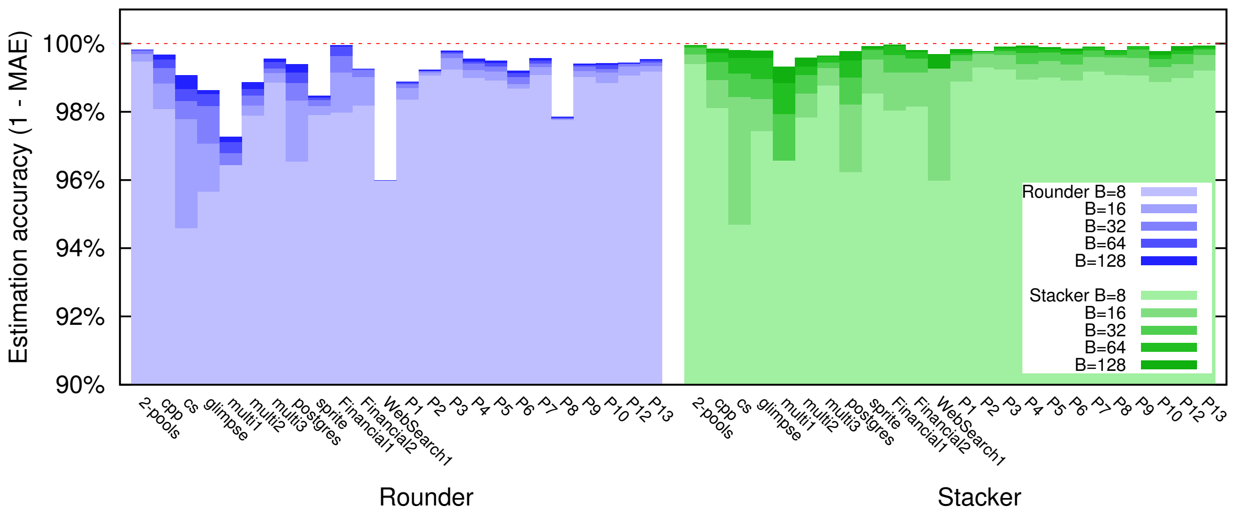


Figure 6: ***Accuracy on cache traces.*** *Comparison of prediction accuracy, measured by mean average error (MAE) between* MIMIR*'s predicted and optimal* LRU *hit rate curves, across canonical cache workloads as we vary the number of buckets B in* ROUNDER *and* STACKER*. Note that the y-axis begins at* 90% *accuracy.*

Table 1: ***Traces.*** *Workloads used to measure accuracy of HRC algorithms, number of requests and configured cache size [27, 28, 35].*

Trace	**2_pools**	**cpp**	**cs**	**glimpse**	**multi1**	**multi2**	**multi3**	**postgres**	**sprite**	**Financial1**	**Financial2**	**WebSearch1**
Requests	100K	9K	7K	6K	16K	26K	30K	10K	134K	1M	3M	1M
Cache size	450	900	1K	3K	2K	3K	4K	3K	1K	50K	50K	50K
Trace	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**	**P9**	**P10**	**P12**	**P13**
Requests	3M	666K	239K	967K	1M	520K	751K	2M	682K	1M	547K	1M
Cache size	50K	50K	50K	50K	50K	50K	50K	50K	50K	50K	50K	50K

trace even as more buckets were added, with accuracy just below 96%. The elements in the trace have large stack distances and thus the workload is best served by a large cache. The ROUNDER approximation falls short of properly balancing the buckets due to large miss rates.

Theorem 1 in §3.6 renders an upper bound on accuracy in terms of average bucket size of items that were hit during the execution of the algorithm. The upper bound is within 5× of the MAE experienced by both algorithms. The large factor is explained by the HRC predictions being nearly accurate, since the average accuracy exceeds 99.8%.

Other stack replacement policies. Most LRU implementations suffer from lock contention on the list head, which needs to be updated on every cache reference. Several real-world systems instead opt to use approximations to LRU such as the CLOCK algorithm [13]. For example, a recent paper of Fan *et al.* showed how the throughput of memcached could be improved through the use of CLOCK and concurrent Cuckoo hashing [19].

In CLOCK, cache items are arranged in a circular linked list, and a "recently-used" activity bit is set when an item is accessed. On a miss, a "hand" steps clockwise through the linked list and resets the activity bits from entries until an item i with an unset activity bit is found. Item i is then evicted and replaced with the requested item, which has its activity bit set, and the clock advances to the following item on the list. The "second chance" aspect of CLOCK provides a 1-bit approximation to the LRU policy. However, CLOCK is not a stack algorithm [13].

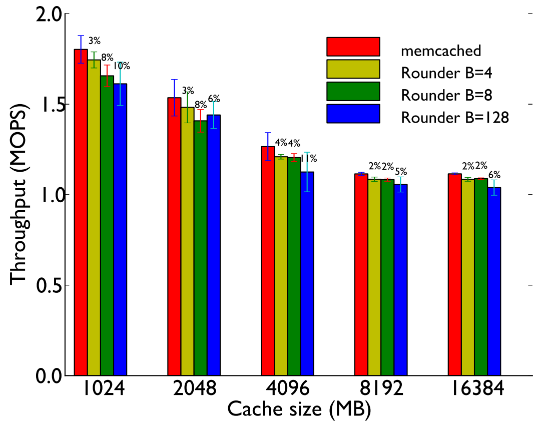
Since both CLOCK and our algorithms are designed to approximate LRU cache evictions, we investigated how well the HRC computed by our methods predict the behavior of CLOCK. We compare against a profiling algorithm SC2 explicitly designed to approximate HRC for CLOCK [11].

We computed the MAE of STACKER compared to the optimal CLOCK HRC, which was computed at every cache size. The average accuracy over the traces in Table 1 ranged from 98.9% for $B = 8$ to 99.3% for $B = 128$ for both aging policies. Although our methods were not designed for CLOCK, they approximated the HRC with significantly higher fidelity than the previous method SC2 [11], whose prediction accuracy averaged 96.4% on the traces.

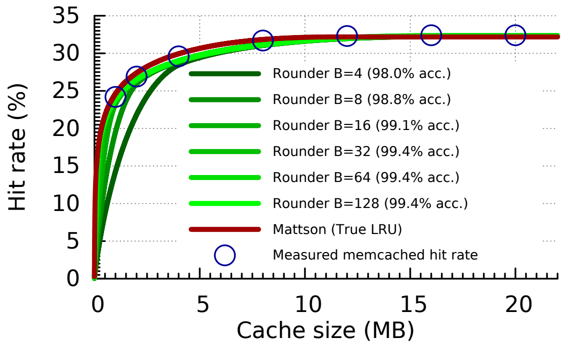
Takeaways. MIMIR *consistently profiles* LRU *hit rate curves with at least* 96% *accuracy, exceeding* 99.8% *when* 128 *buckets are used.* MIMIR *also estimates* CLOCK *hit rate curves with over* 98.9% *accuracy, out-competing approaches designed for profiling* CLOCK *[11].*

4.2 Experiments

To validate our methods on real systems, we integrated a prototype of ROUNDER within memcached [21].



(a)



(b)

Figure 7: *(7a)* ***Performance****. Throughput of* memcached *with and without* MIMIR *with* ROUNDER *with varying number of buckets B over different cache sizes while stress-tested by 10* libmemcached *clients. Error bars represent one sample standard deviation over* 10 *runs. (7b)* ***Accuracy****.* MIMIR *with* ROUNDER *and* MATTSON *running on* memcached *with varying buckets on the YCSB* **b2** *benchmark. The circles are true hit rates reported by* memcached *at various sizes on the same trace. The legend includes accuracy figures.*

Implementation. memcached exposes a lightweight get-set-delete interface to serve key-values tuples from DRAM. Internally, keys and values are stored within *slabs* maintained by a custom memory allocator. Slabs are further subdivided into chunks and all chunks of the same size are said to belong to the same *slab class*. Each slab class runs its own separate LRU queue, implemented as a doubly-linked list. Memcached is multi-threaded and uses global locks on index updates and cache eviction. The source code for recent optimizations within memcached [19] was not available.

We modified memcached 1.4.20 to use the MIMIR interface with the ROUNDER aging policy on cache accesses, and hooked it up to our C implementation of MIMIR. The per-element space overhead in our MIMIR prototype is four bytes. The changes to the memcached code were minimal.

Platform. We ran our experiments on 11 IBM HS22 blades in an IBM BladeCenter H. Each node in the cluster has 6 Intel Xeon E5645 quad-core CPUs@2.4 GHz with CPU frequency scaling turned off, and 48GB of DRAM. The nodes communicate via 40Gbps QDR Infiniband interconnect using Ethernet, and have shared storage.

Throughput and latency. We measured the request throughput of memcached with and without the MIMIR profiler. We ran both versions on a node in our cluster with 4 concurrent threads, and varied the memory capacity. We deployed libmemcached 1.0.18 on 10 nodes and used the memaslap workload generator. Each node requests 2 million random 16 byte unique keys and 32 byte values via 10 threads. The proportion of `GET` requests to `SET` is 9:1, and we stack 100 `GET`s in a single `MULTI-GET` request. Our experimental set-up follows Fan *et al.* [19] and the value lengths are guided by measurements at Facebook [3].

Figure 7a shows the total throughput from the clients to the central memcached server, each point measuring 2 minutes. The throughput of our augmented memcached is on average 3.6% lower than the original memcached implementation with $B = 4$ buckets, and 8.8% with $B = 128$ buckets that produces more granular statistics. At larger memory sizes when fewer misses need to be handled, throughput degradation is between $2-3\%$ for $B = 8$. We observe a paradoxical drop in throughput for the service as cache size increases. This stems from memory management inefficiencies and coarse-grained lock contention within memcached [19, 45] as the hit rate rises from $\sim 40\%$ at 1GB to effectively 100% at 4GB and up.

The per-request latency overhead from MIMIR's profiling is about 4.9% on average (11.7% at 90th-percentile) for $B = 4$. For $B = 128$, the average latency overhead has reduced to 2% (7.6% at 90th-percentile).

Accuracy. We next measure the prediction accuracy with our MIMIR memcached implementation with larger cache sizes than afforded by our trace-driven simulator. We used the YCSB benchmark [12] to generate workload **b2**, representative of a real-world read-heavy photo tagging site with a heavy-tailed popularity distribution on the items. The overhead of MATTSON prevented us from computing the true LRU HRC for large cache sizes. We restrict ourselves to allocation sizes that allow prediction error to be quantified.

Figure 7b shows the HRC generated by MIMIR on memcached using ROUNDER on a 20MB cache. We also ran MATTSON on the same trace to find a baseline against which to measure accuracy. Finally, we reran memcached at varying cache sizes for the same input to verify that predictions match reality. We plot the hit rate measured by memcached in the circles on the figure. Some discrepancy is expected since the LRU implementation in memcached takes shortcuts to overcome rare performance bottlenecks. The prediction accuracy exceeds 98%, reaching 99.4% with 128 buckets.

Ghost filters. Since probabilistic sets such as counting Bloom filters incur false positives, they can affect prediction accuracy. In a microbenchmark on **multi1**, a challenging trace, we found that LRU with a list of 3 ghost filters had estimation fidelity of 95.5% for caches of up to double the cache size. Our current prototype has between 2-17% throughput overhead from the ghost filters depending on miss rate. In ongoing work, we seek to reduce the overhead through sampling [8] and improving concurrency control.

Takeaways. *Profiling memcached with* MIMIR *and the* ROUNDER *aging policy with $B = 4$ has minor impact on latency and throughput ($2-5\%$) while the HRC estimates at different cache sizes are at least 98% accurate.*

5. Related Work

The challenge of provisioning caches is a recent topic of interest. Hwang and Wood described an adaptive system for optimizing costs of memory caches, but with a main focus is on load balancing through hash rebalancing which is complementary to our approach [26]. Zhu *et al.* [18] argued that adaptively scaling down the caching tier could reduce costs if one can compute the minimum hit rate needed to maintain response time SLAs. The data may be derived directly from the HRC dynamically provided by MIMIR.

Hit rate curves have been discussed to model memory demands since the advent of cache replacement algorithms [14, 31, 34]. Offline evaluation of HRC to help accurately profile programs and analyze patterns in the control flow for locality optimizations has been studied in a series of papers [2, 7, 15]. They achieve high space and time efficiency for profiling the entire HRC of a program, but assume unbounded memory for storing elements rather than a cache-like data structure. The methods rely on self-balancing tree data structures that would make them susceptible to lock contention when subjected to dynamic settings with multiple competing threads, as discussed earlier. In contrast, our methods below have constant overhead per request, assuming the number of buckets is kept constant.

Dynamic evaluation of HRC has been explored in various contexts, from improving memory utilization across virtual machines [23, 29, 33, 47], sharing dynamic memory between applications on the same system [31, 49] to supporting garbage-collected applications [46]. All of these examples exploit context-specific assumptions and optimizations, such as specialized hardware and lack of thread contention in the kernel. The VMWare ESX server samples pages that are utilized to approximate global memory utilization [44], but does not estimate performance at cache size beyond the current allocation. Geiger monitors memory pressure to infer page faults from I/O and employs ghost lists, but their MemRx estimation algorithm traverses the entire LRU list on evictions. Zhao *et al.* [48] couple an LRU list with an AVL-tree for reuse distance lookups and discuss how the overhead can be reduced by disabling the monitor during stable memory accesses. While bursty memory access patterns are common in operating systems and virtual machines, memory caches face continuous request load [3].

Kim *et al.* [31], and later systems in cache architecture such as RapidMRC [42] and PATH [4], partition the LRU list into groups to reduce cost of maintaining distances, which is conceptually similar to our approach except the group sizes are fixed as powers of two. Our variable sized buckets afford higher accuracy in trade for more overhead.

Recently proposed cache replacement policies, such as ARC [35], LIRS [27], CAR [5] and Clock-Pro [28] that are slowly being adopted, open up new questions for monitoring hit rate curves. They are not stack algorithms so our hit rate estimation methods will need to be adapted. The MIMIR algorithm could possibly be generalized for ARC by ARC is internally composed of two stack algorithms, LRU and LFU.

6. Conclusions

Popular websites regularly deploy and use large in-memory caches to enhance scalability and response time of their web sites. Such caches have a price tag, begging the question of how well these resources are being spent. We argue that exposing dynamic curves of cache hit rate versus space (HRCs) allows operators to profile and calibrate their cache resources by understanding how provisioning different capacities to their servers affects both performance and cost.

We introduced MIMIR, a lightweight monitoring system for dynamically estimating HRCs of live cache servers. A key component of MIMIR is a novel approximation algorithm, which partitions the LRU stack into a fixed number of buckets for tracking LRU stack distances with low space and time overhead, and provably bounded average error.

Our experiments on a variety of cache traces show that the HRCs generated by MIMIR are between 96-100% accurate and that the accuracy of our methods can be traded off for time and space overhead by adjusting the number of buckets employed by the estimation algorithm.

To demonstrate practicality of our approach, we plugged MIMIR into the popular memcached system and ran experiments on standard benchmarks. Our measurements indicate that HRCs can be gathered within real systems with high accuracy (over 98%) while incurring a minor drop in the request throughput and latency. We believe accurate performance monitoring of live distributed caches is feasible in practice, and can be a valuable tool to facilitate elastic and efficient provisioning of cache capacity.

Acknowledgments

We thank Björn Þór Jónsson for his valuable help with the paper, Song Jiang for sharing his traces, Frey Alfredsson for assisting with experiments, and our shepherd Qiong Luo, colleagues at Facebook, Twitter and CloudPhysics and the reviewers constructive feedback. Our work was supported in part by a grant-of-excellence (#120032011) from the Icelandic Research Fund and funding from Emory University and Reykjavik University.

References

[1] Redis key-value store. `http://redis.io`.

[2] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. *SIGPLAN Notices*, 38(2 supplement):37–43, June 2002.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *SIGMETRICS '12*, pages 53–64, 2012.

[4] R. Azimi, L. Soares, M. Stumm, T. Walsh, and A. D. Brown. PATH: page access tracking to improve memory management. In *ISMM '07*, pages 31–42, 2007.

[5] S. Bansal and D. S. Modha. CAR: Clock with Adaptive Replacement. pages 187–200, 2004.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

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

[8] E. Berg and E. Hagersten. StatCache: A probabilistic approach to efficient and accurate data locality analysis. In *Proceedings of the 2004 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '04, pages 20–27, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7803-8385-0.

[9] N. G. Bronson, J. Casper, H. Chafi, and K. Olukotun. A practical concurrent binary search tree. *SIGPLAN Not.*, 45 (5):257–268, Jan. 2010.

[10] G. Chockler, G. Laden, and Y. Vigfusson. Data caching as a cloud service. In *LADIS '10*, pages 18–21, 2010.

[11] G. Chockler, G. Laden, and Y. Vigfusson. Design and implementation of caching services in the cloud. *IBM Journal of Research and Development*, 55(6):9:1–9:11, 2011.

[12] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC '10*, pages 143–154, 2010.

[13] F. J. Corbato. *Festschrift: In Honor of P. M. Morse*, chapter A Paging Experiment with the Multics System, pages 217–228. MIT Press, 1969.

[14] P. J. Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, 6(1):64–84, Jan. 1980.

[15] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *PLDI '03*, pages 245–257, 2003.

[16] A. Dragojević, D. Narayanan, O. Hodson, and M. Castro. FaRM: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 401–414, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-09-6.

[17] M. R. Ebling, L. B. Mummert, and D. C. Steere. Overcoming the network bottleneck in mobile computing. In *WMCSA '94*, pages 34–36, 1994.

[18] B. Fan, H. Lim, D. G. Andersen, and M. Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 23:1–23:12, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. .

[19] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and concurrent MemCache with dumber caching and smarter hashing. In *NSDI '13*, pages 385–398, 2013.

[20] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Trans. Netw.*, 8(3):281–293, June 2000. ISSN 1063-6692. .

[21] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, (124):72–74, 2004.

[22] S. Hart, E. Frachtenberg, and M. Berezecki. Predicting Memcached throughput using simulation and modeling. In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, pages 40:1–40:8, San Diego, CA, USA, 2012. Society for Computer Simulation International. ISBN 978-1-61839-786-7.

[23] M. Hines, A. Gordon, M. Silva, D. Da Silva, K. D. Ryu, and M. Ben-Yehuda. Applications know best: Performance-driven memory overcommit with Ginkgo. In *CloudCom'11*, pages 130–137, 2011.

[24] Y.-J. Hong and M. Thottethodi. Understanding and mitigating the impact of load imbalance in the memory caching tier. In *Proceedings of the 4th ACM Symposium on Cloud Computing*, SOCC '13, pages 13:1–13:17, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. .

[25] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 167–181, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. .

[26] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 33–43, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7.

[27] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, June 2002.

[28] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *ATEC'05*, pages 35–35, 2005.

[29] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS XII*, pages 14–24, 2006.

[30] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM. ISBN 0-89791-888-6. .

[31] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer

management scheme that exploits sequential and looping references. In *OSDI '00*, pages 9–9, 2000.

[32] H. Lim, D. Han, D. G. Andersen, and M. Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 429–444, Seattle, WA, Apr. 2014. USENIX Association. ISBN 978-1-931971-09-6.

[33] Y. Lu, T. Abdelzaher, C. Lu, and G. Tao. An Adaptive Control Framework for QoS Guarantees and its Application to Differentiated Caching Services.

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

[35] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST '03*, pages 115–130, 2003.

[36] R. Nishtala, H. Fugal, S. Grimm, et al. Scaling Memcache at Facebook. In *NSDI '13*, pages 385–398, 2013.

[37] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *SOSP '95*, pages 79–95, 1995.

[38] M. Rajashekhar and Y. Yue. Caching with twemcache. `http://engineering.twitter.com/2012/07/caching-with-twemcache.html`.

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

[40] A. J. Storm, C. Garcia-Arellano, S. S. Lightstone, Y. Diao, and M. Surendra. Adaptive self-tuning memory in DB2. In *VLDB '06*, pages 1081–1092, 2006.

[41] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Cache Partitioning for Simultaneous Multithreading Systems. In *PDCS'01*, pages 116–127, 2001.

[42] D. K. Tam, R. Azimi, L. B. Soares, and M. Stumm. RapidMRC: approximating L2 miss rate curves on commodity systems for online optimizations. In *ASPLOS XIV*, pages 121–132, 2009.

[43] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, J. Hoon, S. Kulkarni, N. Lawrence, M. Marchukov, D. Petrov, and L. Puzar. TAO: How Facebook serves the social graph. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 791–792, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. .

[44] C. A. Waldspurger. Memory resource management in VMware ESX server. In *OSDI '02*, 2002.

[45] A. Wiggins and J. Langstone. Enhancing the scalability of memcached. `http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached-0`.

[46] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. CRAMM: virtual memory support for garbage-collected applications. In *OSDI '06*, pages 103–116, 2006.

[47] W. Zhao and Z. Wang. Dynamic memory balancing for virtual machines. In *VEE '09*, pages 21–30, 2009.

[48] W. Zhao, X. Jin, Z. Wang, X. Wang, Y. Luo, and X. Li. Low cost working set size tracking. In *ATC'11*, pages 17–23, 2011.

[49] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *ASPLOS XI*, pages 177–188, 2004.

[50] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch. Saving cash by using less cache. In *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*, HotCloud'12, Berkeley, CA, USA, 2012. USENIX Association.