2021

Efficient Modeling of Random Sampling-Based LRU Cache

Junyao Yang
Michigan Technological University, junyaoy@mtu.edu

EFFICIENT MODELING OF RANDOM SAMPLING-BASED LRU CACHE

By
Junyao Yang

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

MICHIGAN TECHNOLOGICAL UNIVERSITY
2021

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

Department of Computer Science

Thesis Advisor:	*Dr. Zhenlin Wang*
Committee Member:	*Dr. Soner Onder*
Committee Member:	*Dr. Junqiao Qiu*
Committee Member:	*Dr. Kui Zhang*
Department Chair:	*Dr. Linda Ott*

Contents

List of Figures

List of Tables

Preface

This thesis work contains material previously published in the Proceedings of the 50th International Conference on Parallel Processing (ICPP '21) [29]:

> Junyao Yang, Yuchen Wang, and Zhenlin Wang. 2021. Efficient Modeling of Random Sampling-Based LRU. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 32, 11 pages. https://doi.org/10.1145/3472456.3472514

Abstract

The Miss Ratio Curve (MRC) is an important metric and effective tool for caching system performance prediction and optimization. Since the Least Recently Used (LRU) replacement policy is the de facto policy for many existing caching systems, most previous studies on efficient MRC construction are predominantly focused on the LRU replacement policy. Recently, the random sampling-based replacement mechanism, as opposed to replacement relying on the rigid LRU data structure, gains more popularity due to its lightweight and flexibility. To approximate LRU, at replacement times, the system randomly selects K objects and replaces the least recently used object among the sample. Redis implements this approximated LRU policy. We observe that there can exist a significant miss ratio gap between exact LRU and random sampling-based LRU under different sampling size K; therefore existing LRU MRC construction techniques cannot be directly applied to random sampling based LRU cache without loss of accuracy.

In this thesis, we present a new probabilistic stack algorithm named KRR which can be used to accurately model random sampling based-LRU cache with arbitrary sampling size K. We propose two efficient stack update algorithms which reduce the expected running time of KRR from $O(N * M)$ to $O(N * log^2 M)$ and $O(N * logM)$, respectively, where N is the workload length and M is the number of distinct objects. Our implementation generates accurate miss ratio curves for both fixed and variable block size cache. Furthermore, we adopt spatial sampling which further reduces the running time of KRR by several orders of magnitude, and thus enables practical, low overhead online application of KRR.

Chapter 1

Introduction

Cache has always been one of the most critical layers in the memory hierarchy. Modern high-performance systems rely on caching to reduce data transfer latency and achieve high throughput. For large-scale web services, key-value caches like Redis and Memcached [13, 16] are crucial for ensuring low-latency service when serving

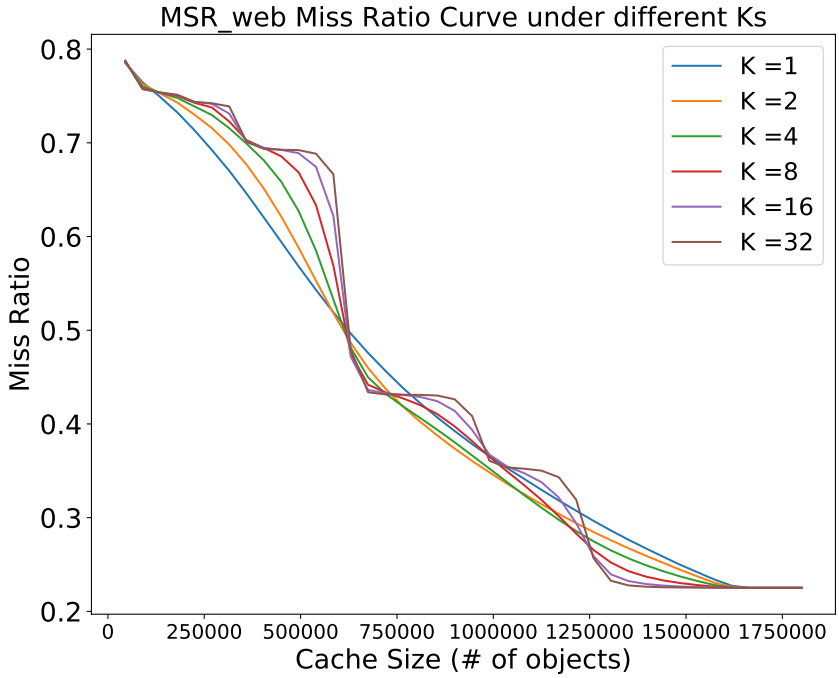


Figure 1.1: MRCs of MSR Web with K-LRU

enormous workloads. Cache design has been studied for decades. One vital component in cache design is the cache replacement policy. There are many existing advanced replacement policies such as ARC [15], MultiQueue [33] and CACHEUS [21]. Although these algorithms work well for most workloads, they all have similar downsides. First, these advance algorithms require additional sorted data structures to maintain objects' relative ordering. As a result, the cache must spend extra time and space in maintaining the ordering of cached objects. Second, these data structures are often very rigid in nature, in other words, once the replacement rule is fixed, it is hard to reconfigure the replacement rules dynamically due to the nature of the data structures.

To avoid expensive ordering data structures, many existing schemes have adopted the idea of random sampling: On eviction, cache randomly selects a small number of items and then evicts item with the lowest priority. Ideally, the evicted item from a set of relatively small random sampled items could closely approximate the lowest priority in the whole cache [8]. The commercial in-memory cache, Redis, implements both of its LRU and LFU replacement schemes based on such random sampling approach [13], and they have demonstrated that with a relatively small sampling size (10), random sampling-based LRU closely approximates true LRU. For simplicity, we use K-LRU to denote random sampling-based LRU policy, where K represents cache's eviction sampling size. Two recent function-based cache replacement schemes, Hyperbolic caching for Redis [8] and LHD for Memcached [3], also rely on the random sampling technique to relax the expensive overhead related to maintaining all object's ranking. By removing the rigid ordering data structure, a random sampling caching scheme also provides great flexibility. First, one can dynamically change cache's priority function online to adapt change in workload patterns. Second, one can dynamically configure the sampling size of random sampling. Wang et al. show that different sampling sizes impose a large impact on cache's miss ratio (Figure 1.1) [25]. By dynamically configuring the sampling size of random sampling-based LRU, they proposed DLRU which can always outperform fixed sampling size cache. Motivated by the impact of eviction sampling size on cache's miss ratio, this thesis aims at accurately modeling cache under K-LRU policy for arbitrary sampling size K.

A Miss Ratio Curve (Figure 1.1), or MRC, is a function mapping from cache sizes to miss ratios. It is an extremely useful tool for cache memory management [10, 11, 24, 26]. Unfortunately, as of today, most studies on efficient MRC construction are focused on the rigid data structure-based LRU cache [6, 9, 11, 24, 27]. Their approaches are derived from Mattson et al's LRU *stack algorithm* which can construct an MRC through one pass of accesses [14]. Years of efforts have improved the asymptotic complexity of the LRU stack algorithm from $O(NM)$ to $O(N)$ where N is the trace length and M is the number of distinct objects. However, the linear time MRC algorithms can lose accuracy. In this paper, we focus on modeling K-LRU.

As shown in Figure 1.1, with random sampling-based LRU (K-LRU) cache, different sampling sizes could have a huge impact on cache's miss ratio. Existing LRU MRC construction techniques are no longer suitable for a cache with the K-LRU policy. We propose a new efficient stack algorithm, which can be used to construct K-LRU MRC with arbitrary K. Here we summarize our major contributions as following:

1. To correctly model the behavior of K-LRU cache, we present a new probabilistic stack algorithm, *KRR*, which statistically approximates the K-LRU policy with arbitrary K. When K is relatively large, KRR closely approximates the LRU policy. When $K = 1$, KRR degenerates to Mattson's RR stack algorithm, a stack algorithm which is statistically equivalent to the random replacement policy.

2. We propose two efficient stack update mechanisms which reduce KRR's expected running time from $O(NM)$ to $O(Nlog^2M)$ and $O(NlogM)$, respectively. Together with the spatial sampling technique proposed by Waldspurger et al. [24], we further reduce the time overhead to an extremely small magnitude which makes it practical for constructing a K-LRU MRC online.

3. We evaluate the accuracy of KRR stack algorithm using MSR, YCSB, Twitter workloads [1, 2, 30]. By comparing with existing MRC techniques, we show that KRR yields a highly accurate MRC for K-LRU cache with low space and time overhead.

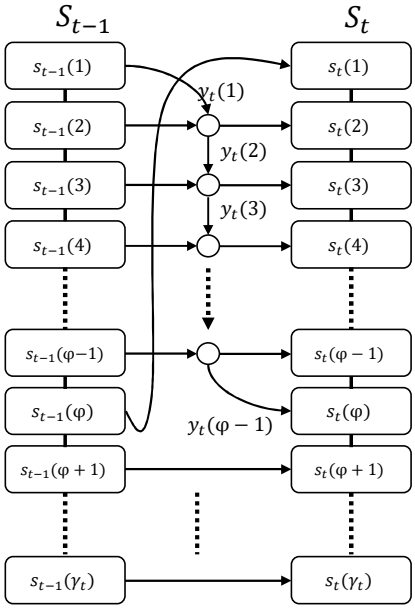
Chapter 2

Background

In this chapter, we first briefly describe the evolution of MRC construction techniques. Then, we will show details about the first single-pass MRC construction algorithm, namely Mattson's generic stack algorithm and introduce necessary notations that will be used in later chapters along the way. Next, we address motivation and challenges on K-LRU stack distance analysis. Lastly, we describe the spatial sampling technique adopted from SHARDS [24].

2.1 Miss Ratio Curve

A Miss Ratio Curve relates miss ratio to cache size. Given the MRC of a workload, one can immediately know the miss ratio for any cache allocation. The MRC is thus a very useful tool for cache memory management, ranging from hardware caches, Java heap management, to in-memory key-value stores, to name a few [10, 20, 28, 31]. In early 1970s, Mattson *et al.* [14] introduced a generalized stack algorithm that models a general class of replacement policies that satisfy the inclusion property (see Section 2.2). The algorithm models the cache as a stack, and the stack location i (stack top location = 1), where the referenced object resides, is called the object's *stack distance* (to the stack top). Under the stack model, an MRC can be calculated based on *stack distance distribution*: the miss ratio of a cache size c is the probability of stack distance greater than c. Since then, a rich set of of studies have been focused on developing an efficient stack distance model for the LRU policy. Olken *et al.* [17] reduced the algorithm complexity down to $O(NlogM)$ by replacing the linear stack structure with the balanced search tree, and till today, $O(NlogM)$ remains to be the



1: **procedure** StackUpdate(S_{t-1}, ϕ)
2: ▷ S: Ordered Stack
3: ▷ ϕ: stack distance of referenced item
4: $y_t(1) \leftarrow S_{t-1}(1)$
5: $S_t(1) \leftarrow S_{t-1}(\phi)$
6: **for** $i \leftarrow 2...\phi - 1$ **do**
7: $S_t(i) \leftarrow maxPriority\,(y_t(i-1), S_{t-1}(i))$
8: $y_t(i) \leftarrow minPriority\,(y_t(i-1), S_{t-1}(i))$
9: **end for**
10: $S_t(\phi) \leftarrow y_t(\phi - 1)$
11: **end procedure**

Figure 2.1: Mattson Stack Update Process [14]

lower bound for generating exact MRCs on an LRU cache. To further reduce the running time, much of the attention has been shifted to stack distance approximation techniques [11, 24, 26] which further reduce the time complexity to super linear or linear with sacrifices in a slight loss of accuracy.

2.2 Mattson's Stack Algorithm

Here we briefly describe the general stack algorithm proposed by Mattson *et al.* In general, a replacement algorithm is called a stack algorithm if such replacement algorithm satisfies the inclusion property, that is, $B_t(C) \subset B_t(C+1)$, where $B_t(C)$ is a set of distinct objects in a cache of arbitrary size C at given time t. The inclusion property of stack algorithm makes it possible to generate an MRC in just one pass of the trace which motivates an efficient stack model for the K-LRU replacement policy. Under the general stack model, all previously referenced objects have an associated priority. Depending on the replacement policy, the object's priority can change over time. At any given time, all referenced object's priorities form a total ordered set. Mattson et al. show that in order to preserve the inclusion property, the stack S_t at time t must be maintained according to following constraints:

$$S_t(1) = x_t \tag{2.1a}$$
$$S_t(i) = maxPriority\left(y_t(i-1), s_{t-1}(i)\right) \qquad \text{for } 2 \leq i < \phi \tag{2.1b}$$
$$S_t(\phi) = y_t(\phi - 1) \tag{2.1c}$$
$$S_t(j) = S_{t-1}(j) \qquad \text{for } \phi < j \leq \gamma_{t-1} \tag{2.1d}$$

where:

x_t : object referenced at time t
$S_t(i)$: object at i_{th} stack position at time t.
$y_t(i)$: the lowest priority object in cache of capacity i at time t.
γ_t : total distinct referenced objects at time t
ϕ : x_t's stack distance[1], if x_t never referenced, $\phi = \gamma_t$

The $maxPriority()$ function in stack maintenance procedure above is a function comparing priority of $y_t(i-1)$ and $s_{t-1}(i)$. Intuitively, the lower priority object determined by $maxPriority()$ function is the evicted object in cache of size i. For simplicity, one can think that the only difference among stack algorithms is their $maxPriority()$ function. Figure 2.1 illustrates a general stack update process, which typically takes linear time, on average, with respect to the stack size. Given an access stream, $X = x_1, x_2, ..., x_t$, one can obtain a stack distance histogram (SDH)

[1]stack distance = $sd(S_t(i)) = i$

by processing the access stream via the corresponding stack algorithm. For an LRU stack, the stack update process is particularly trivial; Since objects' priority ordering is equivalent to stack ordering in the LRU stack, then, on a stack update, all objects from stack position 1 to $\phi - 1$ are push down by one position, or equivalently it takes $O(1)$ to move the referenced object to stack top when the stack is organized as a doubly-linked list. However, finding the stack distance of an object still takes expected linear time with respect to the stack size. Mattson's LRU stack algorithm is thus $O(NM)$.

2.3 Motivation and Challenges

As demonstrated by Figure 1.1, the cache can have a very different miss ratio under K-LRU when K varies. It is desirable to have an efficient model to construct an MRC for K-LRU. Current stack distance approximation techniques such as AET, Counterstack, and SHARDS[2] only model stack distance distribution for caches under the exact LRU policy. They clearly are not the best choice for a K-LRU cache. To tackle this problem, we propose a new MRC construction method that models K-LRU's miss ratio under arbitrary K and cache size.

A stack model is attractive in that it can generate an MRC in one pass. However, there are two main challenges to develop a stack algorithm for K-LRU. First, the stack algorithm must satisfy the inclusion property. It's easy to check that probabilistic replacement strategies like K-LRU does not satisfy the inclusion property. This makes it impossible to directly perform stack distance analysis on the K-LRU policy. To circumvent it, we defined a new stack algorithm, *KRR*, which statistically approximates the K-LRU policy. In this way, predicting K-LRU's miss ratio is equivalent to predicting KRR's miss ratio. Second, the original stack algorithm has a linear stack search/update time cost, which is impractical for online usage. To overcome such high cost, we introduce a new stack update procedure in Section 4.3.2 for KRR which only requires $O(logM)$ time overhead per stack update.

The original stack model was designed to model a class of replacement algorithms under the assumption that the size of objects is fixed. This assumption works for hardware cache where the size of a cache block is fixed. However, this assumption does not always hold for software cache. Recent studies [5, 30] show that the size of objects in the in-memory cache can be very diverse, and the size distribution of workloads is

[2]The SHARDS here is specifically refer to spatially scaled down version of LRU balanced tree, not the spatial sampling method.

usually not static over time. Moreover, Pan *et al.* demonstrate that miss ratio curves constructed under uniform size assumption can significantly deviate from the true miss ratio curve when the workloads follow non-uniform size distribution [18]. Thus, our last challenge is to extend the KRR stack algorithm to handle workloads with variable object sizes. To handle variable object sizes, we must change the granularity of stack distance from object to byte. In Section 4.4.1, we show that the cumulative size distribution along the stack can be captured by adding a simple mechanism on top of the KRR stack which allows us to accurately approximate stack distance in byte-level granularity.

2.4 Spatial Sampling

For any stack algorithms, an MRC can be calculated from the generated stack distance histogram. The problem is that it is very expensive, in both space and time, to obtain the actual SDH for a long trace because the asymptotic space/time cost of the stack algorithm is correlated with the number of unique references in the workload, which can be very large. Due to the large overhead, it is impractical to directly use stack algorithm online. In order to make it suitable for online usage, we adopt the uniformly random spatial sampling technique described in SHARDS [24]. Instead of feeding entire reference streams to the stack model, spatial sampling technique uses the sampling condition $hash(L)\ mod\ P < T$, with referenced key L, modulus P and threshold T, to collect only a subset of references. Ideally, the effective sampling rate is $R = T/P$. As shown by Waldspurger *et al.*, for majority of workloads tested, the sampled subset has very high statistical similarity compared to the original workload, even with R = 0.001. By combining such spatial sampling technique together with our fast stack update algorithm (Section 4.3), we show that our algorithm can be efficient enough for online MRC prediction.

Chapter 3

Random Sampling Based Replacement and K-LRU

In this chapter, we continue to set up some notations and formalize more details about the K-LRU policy.

An object x's recency r can be defined as $r(x) = \frac{1}{\text{time since last referenced}}$. Under the LRU policy, all objects are ranked according to their recency and the least recently used object will be remove from cache on eviction. The cache with capacity C can be described as a total ordered set $\{x_d : 1 \leq d \leq C\}$ where x_1 is the object with highest ranking, that is, the object most unlikely to be evicted. We define $\rho_{t,C}(r) : r \to d$ as the mapping function that maps object's recency r to object's relative priority ranking d in cache of size C at time t.

There are two versions of random sampling-based cache. On eviction, when sampling K objects from the cache, sampling can be done with or without "placing back" the sampled objects. With placing back, a sampled object can be sampled again, although the probability is small given a small K and a large C. Existing implementation as used in Redis adopts placing-back sampling [13]. For consistency, in the remaining sections, we assume K-LRU is implemented using "placing back" sampling. However, our proposed solution can be similarly applied to the K-LRU sampling policy without placing back through a few tweaks.

Proposition 1. *In a random sampling-based (with placing back) cache with cache size C and sampling size K, the eviction probability, $Q_{C,K}(x == x_d)$, of the object x_d with ranking d is:*

$$Q_{C,K}(x == x_d) = \frac{d^K - (d-1)^K}{C^K} \tag{3.1}$$

Proof. For simplicity, assume that, on an eviction from a cache of size C, the objects' ranks are from 1 to C. Now K-LRU eviction is equivalent to randomly selecting K integers from the set $\{1...C\}$ and choose the largest value, d, from the selected integers. With placing back, d can appear 1 to K times. If d appears i times, there are $\binom{K}{i}$ ways to choose d. For d is the largest integer among the selected K integers, the remaining $K-i$ integers must be smaller than d. There are only $(d-1)^{K-i}$ ways to select the remaining integers. Together, there are $\sum_{i=1}^{K}\binom{K}{i}(d-1)^{K-i}$ ways where d is the largest among selected k integers. Next, simplifying the expression, we get $d^K-(d-1)^K$. Hence, $Q_{C,K}(x == x_d) = \frac{d^K-(d-1)^K}{C^K}$. □

Proposition 2. *In a random sampling-based (without placing back) cache with cache size C and sampling size K, the eviction probability, $Q_{C,K}(x == x_d)$, of the object x_d with ranking d is:*

$$Q_{C,K}(x == x_d) = \begin{cases} 0 & if\ d < K \\ \frac{K(d-1)!(C-K)!}{(d-K)!C!} & otherwise \end{cases} \tag{3.2}$$

Proof. Without placing back, obviously the objects with rank smaller than K will never be evicted. Then given an object with rank d, where $K \leq d \leq C$, the eviction probability of such object is equivalent to the probability of select remaining $K-1$ objects such that the rank of these objects is strictly less than d. There are $\binom{d-1}{K-1}$ ways to select these $K-1$ objects, and $\binom{C}{K}$ ways to select K objects out of C objects. Hence, the probability $Q_{C,K}(x == x_d) = \frac{\binom{d-1}{K-1}}{\binom{C}{K}} = \frac{K(d-1)!(C-K)!}{(d-K)!C!}$ □

One can check that under relative small K and large cache size, these two versions yield approximately the same eviction probability. From Proposition 1, we see that an object with low ranking (larger d) has a higher chance of been evicted in random sampling based cache.

Now, we formulate K-LRU as a probabilistic policy:

Definition 1. *Replacement policy K-LRU is a probabilistic policy such that, on cache eviction, the eviction probability of the object with recency r is $Q_{C,K}(x == x_{\rho(r)})$.*

Chapter 4

The KRR Stack Algorithm

In this chapter we extend original stack processing techniques to handle the non-stack algorithm K-LRU. To address the first challenge where K-LRU is not a stack algorithm, we define a new stack algorithm called KRR. We show that, under a coarse-grained assumption on object's recency, K-LRU and KRR behave statistically the same. Next, to overcome the linear overhead on a stack update, we introduce two new fast stack update approaches that reduce update complexity from linear to $O(log^2 M)$ and $O(logM)$, respectively. To handle variable object sizes, we describe a simple mechanism on top of the KRR stack which captures byte-level size cumulative distribution along the stack while maintains the overall asymptotic complexity of the original KRR stack algorithm. Lastly, the last section of this chapter sets out few implementation details.

4.1 KRR Modeling

Intuitively, to construct a new stack algorithm that is statistically equivalent to K-LRU, the new stack algorithm must ensure the eviction probability in Definition 1. That is, the eviction probability of the object with recency r under such stack algorithm must be equal to $Q_{C,K}(x == \rho(r))$. Unfortunately, there are two problems associated with maintaining the eviction probability $Q_{C,K}(x == \rho(r))$. First, to obtain object $s_t(i)$'s priority ranking $\rho(r)$, one must know the total number of objects that appear before $s_t(i)$ on the stack and have higher recency than $s_t(i)$. This appears to be a very costly task as an intuitive scan of stack for recency ranking would take time in $O(MlogM)$ per update. We would like to avoid it for efficiency purpose.

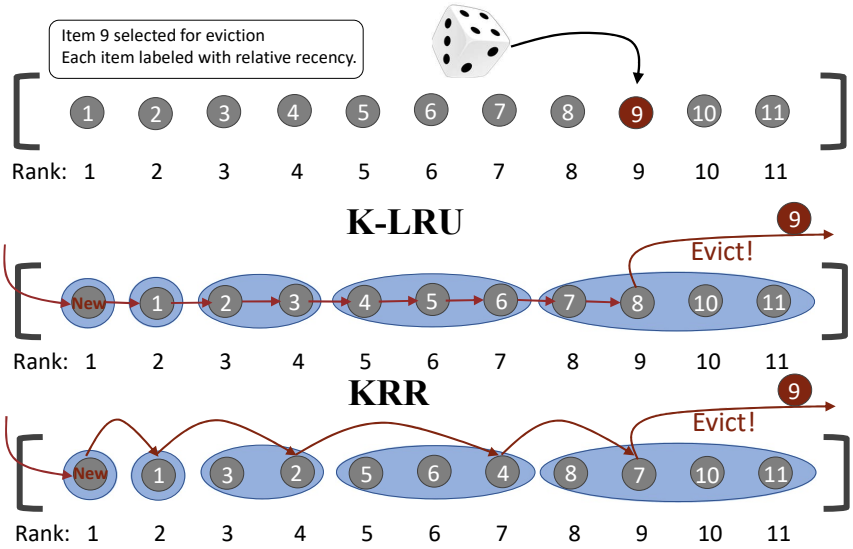


Figure 4.1: Eviction comparison between K-LRU and KRR. The red edge represents movement of objects' ranks. The blue oval groups a coarse-grained ordering of objects' recency.

Second, since object $s_t(i)$'s eviction probability is associated with its relative recency in a cache with size i, the eviction probability is not fixed with respect to its stack position. This is also problematic, as we will see in Section 4.3, an unpredictable eviction probability for a given stack position will make it especially challenging to perform fast stack updates. With the above problems, it is difficult to construct a stack algorithm that is both updates efficient and statistically equivalent to K-LRU. Thus, we proposed an alternative stack algorithm, KRR, which is not perfectly equivalent to but closely approximates the K-LRU policy.

The KRR algorithm is motivated by one simple observation. As illustrated by Figure 2.1, the general stack update is one-way downward shifts of a subset of the stack. The most recently referenced object is always pushed to the top of the stack and then monotonically move downwards. Therefore, the object that appears at the lower stack position is more likely to be less recently referenced than objects that appear above it. In particular, the LRU stack is an extreme case, where all items' recency ordering perfectly matches the stack ordering. In the case of K-LRU, as K increases, the likelihood of the lower stack position object has lower recency increases. Based on such observation, we propose KRR based on an approximation on object $s_t(i)$'s recency:

Assumption 1.
$s_t(i)$ is the least recently used among $\{s_t(j) \mid 1 \leq j \leq i\}$, or equivalently, $\rho_{t,i}(r(s_t(i))) = i$.

Base on above assumption, we now start constructing KRR's *maxPriority* function.

The $maxPriority$ function takes two inputs $s_{t-1}(i)$ and $y_t(i-1)$ then returns the one with higher priority. In other words, object $s_t(i)$ will be replaced by $y_t(i-1)$ if $s_{t-1}(i)$ is evicted from cache of size i at time t. Under Assumption 1 where the object at the i_{th} stack position has relative ranking i in cache of size i, the probability of $s_{t-1}(i)$ being evicted can be calculated, according to Equation 3.2, as $Q_{i,K}(x == x_i) = \frac{i^K-(i-1)^K}{i^K}$. Equivalently, the probability of $s_{t-1}(i)$ staying in cache at time t can be simplified to $\left(\frac{i-1}{i}\right)^K$. Then, the $maxPriority$ function for KRR can be formally described as:

$$maxPriority(y_t(i-1),s_{t-1}(i)) = \begin{cases} s_{t-1}(i) & random(0,1) < \left(\frac{i-1}{i}\right)^k \\ y_t(i-1) & otherwise \end{cases} \quad (4.1)$$

With $maxPriority$ function defined, one can trivially simulate the KRR replacement scheme using Mattson's linear stack update procedure described in Section 2.2.

Under the KRR stack algorithm, as K increases, the probability of $s_{t-1}(i)$ stay in its position decreases. With a large enough K, every $s_{t-1}(i)$ will be replaced by $y_t(i-1)$ which behaves exactly like the LRU stack. When K=1, we see that the KRR stack degenerates to Mattson's RR algorithm [14], which has a stay probability of $\frac{i-1}{i}$ for object $s_{t-1}(i)$. We coined the name "KRR" because this stack algorithm can be considered as an extension of Mattson's RR stack algorithm.

Figure 4.1 illustrates the difference between the KRR and K-LRU replacement algorithms on cache eviction. Both KRR and K-LRU maintain object's ranking, the difference is that K-LRU cache maintains object's ranking implicitly through object's recency, where the more recent object ranks higher and less recent one ranks lower; On the other hand, the KRR replacement policy maintains object's ranking explicitly through stack update procedure under maxPrioty function described above, or we say object's ranking at time t under KRR is exactly object's stack position at time t.

4.2 Correctness

In order to better understand how well KRR approximates K-LRU, we first focus on the eviction probability of an arbitrary object $s_t(i)$ in the KRR cache of size C, $\Phi_{C,K}(s_t(i))$. According to the KRR algorithm, an object on stack position i, denoted as $s_t(i)$, will be evicted from cache of size C, if and only if, objects $y_{t+1}(i-1)$ and $s_t(i+1)$, $s_t(i+2)$, ..., $s_t(C)$ all have higher priority than $s_t(i)$. Mattson *et al* verified that eviction probability of an arbitrary object under RR is equivalent to random

eviction, that is $\Phi_{C,1}(s_t(i)) = \frac{1}{C}$. Using same approach we show that:

$$\begin{aligned}\Phi_{C,K}(s_t(i)) &= \left(\frac{i^K - (i-1)^K}{i^K}\right) * \left(\frac{i}{i+1}\right)^K * \left(\frac{i+1}{i+2}\right)^k * ... \left(\frac{C-1}{C}\right)^K \\ &= \frac{i^K - (i-1)^K}{C^K}\end{aligned} \tag{4.2}$$

Based on Definition 1 and $\Phi_{C,K}(s_t(i))$, we see that KRR and K-LRU cache yield exactly the same eviction probability for an arbitrary object if Assumption 1 holds true. Hence, the accuracy of using the KRR algorithm to approximate K-LRU depends on effectiveness of Assumption 1.

The K-LRU cache ranks objects according to their recency, thus, when the new object enters the cache, all other objects down shift their ranking by one, their relative ranking to one another remains same. Unlike K-LRU, KRR performs one way shifts of object's rank only on a subset of objects as illustrated by Figure 4.1. Although, less recent objects are still likely to be rank lower in KRR cache, due to these probabilistic shifts, objects' ranking in KRR cache does not fully resembles recency ordering as the K-LRU cache does. Since KRR only orders objects according to their recency at a coarse granularity level, a more recently used object could have a higher chance of being evicted compared to a less recently used object. However, in our evaluation, we observe that using KRR's stack ordering to approximate K-LRU's recency ordering is sufficient to yield a very accurate MRC for most cases. The error magnifies only under an occasional circumstance, such as repeatedly access objects with same recency order, i.e. loop pattern. To further reduce the error, we make a simple modification on the KRR algorithm. In general, the K-LRU cache is more likely to evict less recently used objects compare to the KRR cache, because unlike KRR, K-LRU cache ranks objects strictly by their recency. To fix that, we can increase the K in the KRR algorithm, that is, for a K-LRU with sampling size K, we choose a value K' for the corresponding KRR, such that $K' > K$. By using a larger value K', we increase the eviction probability of object with low rank. This effectively offsets KRR's tendency of evicting more recently used objects. In our evaluation, we find that $K' \approx K^{1.4}$ yields a very accurate approximation for K-LRU.

4.3 Fast Stack Update

As described in Section 2.2, the naive stack algorithm requires $O(M)$ update time for every access. In Figure 2.1, a downshift object would need to be compared to and

swap with the objects from the stack top to the recently hit location based on the *maxPriority* function in Equation 4.1. Clearly, $O(M)$ per update is prohibitive for online processing. To overcome the expensive update overhead, we propose two efficient stack update mechanisms, which reduce the overhead from $O(M)$ to $O(log^2 M)$ and $O(logM)$, respectively.

First, we notice that the probability that *maxPriority* function returns $s_{t-1}(i)$ increases as we scan down the stack in Figure 2.1. This suggests that, for every stack update, the object in $s_t(i)$ remains the same as in $s_{t-1}(i)$ for most stack positions, only a small portion of $s_{t-1}(i)$'s are replaced by $y_t(i-1)$'s. For convenience, we now call the stack position i, where $s_{t-1}(i)$ have lower priority than $y_t(i-1)$ as a *swap position*.

Corollary 1. *Let β_{swap} denote total number of swap positions per stack update, then the expectation $E(\beta_{swap})$ is:*

$$E(\beta_{swap}) = O(KlogM)$$

Proof. The probability for the i_{th} stack position to be a swap position is $1 - \left(\frac{i-1}{i}\right)^K$, then the expectation $E(\beta_{swap})$ can be calculated as:

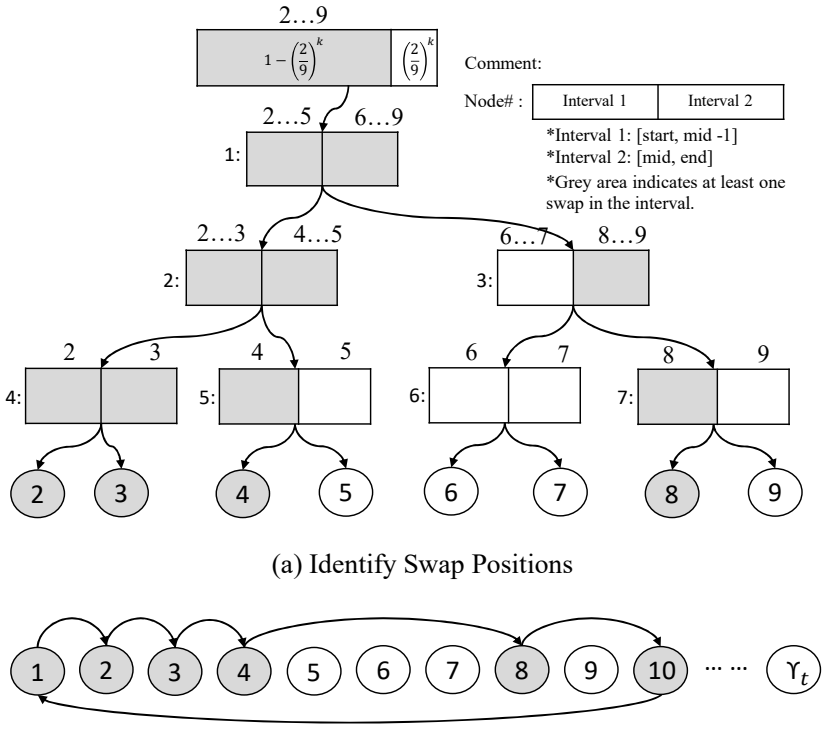
$$\begin{aligned}
E(\beta_{swap}) &= \sum_{x=1}^{\phi-1} \left(1 - \left(\frac{x-1}{x}\right)^K\right), \ 1 \le \phi \le M \\
&\le \int_1^{\phi} \left(1 - \left(\frac{x-1}{x}\right)^K\right) dx \\
&= x \Big|_1^{\phi} - \int_1^{\phi} \frac{(x-1)^K}{x^K}\, dx \\
&= x \Big|_1^{\phi} - \int_1^{\phi} \frac{\sum_{i=0}^{K} \binom{K}{i} x^{K-i}(-1)^i}{x^K}\, dx \\
&= x \Big|_1^{\phi} - \Big(\int_1^{\phi} \frac{\binom{K}{0} x^K (-1)^0}{x^K}\, dx + \int_1^{\phi} \frac{\binom{K}{1} x^{K-1}(-1)^1}{x^K}\, dx \\
&\qquad + \int_1^{\phi} \frac{\binom{K}{2} x^{K-2}(-1)^2}{x^K}\, dx \ + ... \Big) \\
&= x \Big|_1^{\phi} - x \Big|_1^{\phi} + Kln(x) \Big|_1^{\phi} - \Big(\int_1^{\phi} \frac{\binom{k}{2} x^{k-2}(-1)^2}{x^k}\, dx \\
&\qquad + \int_1^{\phi} \frac{\binom{k}{3} x^{k-3}(-1)^3}{x^k}\, dx \, ... \Big)
\end{aligned}$$

$$= Kln(x)\Big|_1^{\phi} - \left(\int_1^{\phi} \frac{\binom{k}{2}x^{k-2}(-1)^2}{x^k}\,dx + \int_1^{\phi} \frac{\binom{k}{3}x^{k-3}(-1)^3}{x^k}\,dx \ldots\right)$$
$$= O(KlogM)$$

□

Based on Corollary 1, with a small constant K, the expected number of swap positions per update is bound by $O(logM)$. Naturally, if all swap positions can be identified prior to the stack update, then update process can be done by simply performing one-way shifts on swap positions from stack top to ϕ, which would be considerably faster than linearly scanning through entire stack.

4.3.1 Approach I: Top Down Stack Update



(a) Identify Swap Positions

(b) Shift corresponding objects

Figure 4.2: Top Down Stack Update Illustration

As mentioned above, fast stack update can be achieved through efficiently simulating all swap positions from stack top to ϕ. Instead of performing random draws on

every position to determine whether it is a swap position, we can recursively divide the problem into smaller sub-problems. First, based on the stack update procedure (Equation 2.1), $s_{t-1}(1)$ and $s_{t-1}(\phi)$ are always swap position, then the task becomes to identify all remaining swap positions between 2 and $\phi - 1$. If there are swap positions between 2 and $\phi - 1$, then we have three different cases: (1) All swap positions are in the interval $\left(2, \lceil \frac{\phi-1}{2} \rceil\right)$. (2) All swap positions are in the interval $\left(\lceil \frac{\phi-1}{2} \rceil + 1, \phi - 1\right)$. (3) Both intervals contains swap positions. In Section 4.1, we specify that the probability that $s_{t-1}(i)$ remains in same stack position i at time t is $\left(\frac{i-1}{i}\right)^K$, which immediately follows that the probability that $s_{t-1}(i)$ to $s_{t-1}(j)$, with $j > i$, all remain in same stack positions at time t is simply $\left(\frac{i-1}{j}\right)^K$. Equivalently, the probability of there is at least one swap position from stack position i to j is $1-\left(\frac{i-1}{j}\right)^K$. With the probability given, we can solve the original problem by breaking it into smaller sub-intervals and recursively solve each sub-intervals. Next, we will demonstrate this top-down strategy by walking through a small example shown in Figure 4.2. The Algorithm 1 is the complete pseudocode for the top down stack update.

Figure 4.2 demonstrates swap positions generation with $\phi = 10$. By definition, position 1 and 10 are swap positions, then we perform a random draw to check whether there are any swap positions in the interval $(2, 9)$. If there are swap positions in the interval, we further break it down into two sub-intervals $(2, 5)$ and $(6, 9)$. Then, we perform the second random draws to determine which sub-intervals contain swap positions. Figure 4.2 indicates that both interval $(2, 5)$ and $(6, 9)$ contain swap positions. We continue the process until all swap positions are identified.

To compute the expected running time of Algorithm 1, we need to sum up the cost of every level in the recursion tree. As an example, Figure 4.2 (a) shows the complete binary state-space tree for the swap position generation between positions 2 to 9. Each node of the complete binary tree represents an interval of consecutive stack positions. On each node, there is an $O(1)$ cost for determining whether the given interval contains swap positions. Note that the recursion tree is not the complete binary tree, we do not traverse nodes that do not contain swap positions. Similarly, we can say the recursion tree is induced by all traversed nodes in the complete binary tree. Thus, the total cost of algorithm 1 is equivalent to the total number of nodes in the recursion tree. In Proposition 3, we show that the expected number of nodes in the recursion tree is bounded by $O(log^2 M)$ if K is a small constant. By using this top-down strategy, we reduce the stack update cost from $O(M)$ to $O(log^2 M)$.

Proposition 3. *Let $\boldsymbol{X}$ denote the total number of nodes traversed during swap positions generation, then the expectation $E(\boldsymbol{X})$ is:*

$$E(\boldsymbol{X}) = O(Klog^2M)$$

Proof.
First, let T denotes the complete binary state-space tree of swap positions generation. Next, we label each node according to the level order traversal of T.

Then, we define indicator random variable X_i such that:

$$X_i = \begin{cases} 1, & \text{if } i^{th} \text{ node is visited.} \\ 0, & \text{otherwise.} \end{cases}$$

Since $\mathbf{X}$ is the total number of node visited, then we have

$$\mathbf{X} = \sum_{i=1}^{|T|} X_i$$

Let $Pr\{i\} = 1 - \left(\frac{\alpha_i - 1}{\beta_i}\right)^K$ denote the probability of the interval associated with i^{th} node has at least one swap position, where α_i is the interval's start position and β_i is the end position.
Then, according to the algorithm, the i^{th} node is visited *iff* the interval (α_i, β_i) contains at least one swap position, therefore, the expected value of X_i can be calculated as:

$$E[X_i] = 1 * Pr\{i\} + 0 * \overline{Pr\{i\}} = 1 - \left(\frac{\alpha - 1}{\beta}\right)^K$$

Immediately follows it, we have

$$E[\mathbf{X}] = E\left[\sum_{i=1}^{|T|} X_i\right] = \sum_{i=1}^{|T|} E[X_i]$$

Next, without loss, we assume the number of positions M is powers of 2. Now we rewrite $E[\mathbf{X}]$ as follow:

$$E[\mathbf{X}] = \sum_{i=1}^{|T|} X_i$$

$$
\begin{aligned}
&= \sum_{i=1}^{|T|} 1 - \left(\frac{\alpha_i - 1}{\beta_i}\right)^K \\
&= \sum_{L=0}^{Log_2(M)-1} \sum_{i=1}^{2^L} 1 - \left(\frac{1 + (i-1)(2^{Log_2(m)-L})}{1 + i * (2^{Log_2(m)-L})}\right)^K \\
&= \sum_{L=0}^{Log_2(M)-1} \sum_{i=0}^{2^L-1} 1 - \left(\frac{2^L + i * M}{2^L + (i+1) * M}\right)^K \\
&\leq \sum_{L=0}^{Log_2(M)-1} \int_0^{2^L} 1 - \left(\frac{2^L + i * M}{2^L + (i+1) * M}\right)^K di \\
&\leq \sum_{L=0}^{Log_2(M)-1} O\left(K * ln\left(\frac{2^L + M * 2^L + 2 * M}{2^L + M}\right)\right) \\
&\leq \sum_{L=0}^{Log_2(M)-1} O\left(K * ln\left(\frac{M^2 + 3M}{2M}\right)\right) \\
&= \sum_{L=0}^{Log_2(M)-1} O\left(K * ln\left(M\right)\right) \\
&= O(K log^2 M)
\end{aligned}
$$

□

Algorithm 1 Approach I: Top Down Stack Update

```
1: procedure StackUpdate(ST, obj)
2:                                   ▷ ST: Ordered Stack, implemented as an arrayList
3:                                   ▷ obj: referenced object
4:                                   ▷ random(): PRNG from [0,1)
5:                                   ▷ new object (cold miss) is attached to the end of the stack before the stack update
6:
7:     if obj.φ == 1 then
8:         return                    ▷ reference obj on the top of the stack, no change needed
9:     end if
10:    if random() > (1/obj.φ)^{ST.k} then
11:        if obj.φ == 2 then        ▷ edge case
12:            add_to_swapArray(2)
13:        else
14:            push_to_unVisitedStack(2, obj.φ)
15:        end if
16:        while unVisitedStack is Not Empty do
17:            elt ← pop_unVisitedStack()
18:            mid ← ⌈(elt.start + elt.end)/2⌉
19:            nsw1 ← ((elt.start − 1)/(mid − 1))^{ST.k}
20:            nsw2 ← ((mid − 1)/elt.end)^{ST.k}   ▷ Probability of no swaps in second interval
21:            sw1 ← 1 − nsw1
22:            sw2 ← 1 − nsw2        ▷ Probability of at least one swap in second interval
23:            ntvl1 ← sw1 ∗ nsw2
24:            ntvl2 ← nsw1 ∗ sw2
25:            wght ← ntvl1 + ntvl2 + (sw1 ∗ sw2)
26:            Rand ← random()
```

```
27:            if Rand < (ntvl1/wght) then                ▷ Fall in region 1
28:                if elt.start == mid − 1 then
29:                    add_to_swapArray(elt.start)
30:                else
31:                    push_to_unVisitedStack(elt.start, mid − 1)
32:                end if
33:            else if Rand < (ntvl1 + ntvl2)/wght then          ▷ Fall in region 2
34:                if elt.end == mid then
35:                    add_to_swapArray(elt.end)
36:                else
37:                    push_to_unVisitedStack(mid, elt.end)
38:                end if
39:            else                                       ▷ Fall in both regions
40:                if elt.end! = mid then
41:                    push_to_unVisitedStack(mid, elt.end)
42:                end if
43:                if elt.start! = mid − 1 then
44:                    push_to_unVisitedStack(elt.start, mid − 1)
45:                end if
46:                if elt.start == mid − 1 then
47:                    add_to_swapArray(elt.start)
48:                end if
49:                if elt.end == mid then
50:                    add_to_swapArray(elt.end)
51:                end if
52:            end if
53:        end while
54:    end if
55:
56:    StackSwaps(ST, obj, swapArray)
57: end procedure
```

4.3.2 Approach II: Backward Stack Update

We now introduce the second stack update method which only requires $O(logM)$ time per update. It is much simpler and only requires about 10 lines of code for stack updates. Mattson et al.'s linear stack update for RR determines swap positions by performing random draws from stack top till $s_{t-1}(\phi)$. We find that a much efficient way can be done by generating swap positions backwards, starting from $s_{t-1}(\phi)$ to

stack top. Let $v_1, v_2, ..., v_\beta, v_{\beta+1}$ denote swap positions ordered by their stack positions in increasing order, where $s_{t-1}(1)$ and $s_{t-1}(\phi)$ are v_1 and $v_{\beta+1}$, respectively. We will start by first identifying swap position v_β. Since v_β is the second to the last swap position, this implies that the objects in stack positions greater than v_β and smaller than ϕ will remain in the same positions at time t. Semantically, the object in swap position v_β is the evicted object in a cache of size $\phi - 1$ at time t. Next, from Equation 4.2, we know the eviction probability of an object in KRR cache is directly associated with its stack position. Furthermore, the cumulative distribution function (CDF) of Equation 4.2 is $P(X \leq x_i) = \left(\frac{i}{C}\right)^K$. Now, we can obtain v_β by simply take the inverse of the CDF with $C = \phi - 1$. For $v_{\beta-1}$, since v_β is already identified, we can compute it using similar idea with $C = v_\beta - 1$. Algorithm 2 shows the complete steps for this backward stack update approach. For total random replacement, or when $K = 1$, this approach degenerates to the D-RAND proposed by Bilardi *et al.*, which is another stack version of random replacement policy [7]. The expected running time for Algorithm 2 is $O(logM)$, because, based on Corollary 1, expected number of swap positions is bound by $O(logM)$, and each iteration of Algorithm 2's while loop computes exactly one swap position. By using Algorithm 2, our KRR model can approximate K-LRU cache in just $O(NlogM)$ time.

Algorithm 2 Approach II: Backward Stack Update

```
1: procedure StackUpdate(ST, obj)
2:                                  ▷ ST: data structure include KRR stack and metadata
3:                                                          ▷ obj: referenced object
4:     i ← obj.φ
5:     while i > 1 do
6:         r ← random()                                  ▷ random(): PRNG from (0,1]
7:         x ← ⌈r^(1/K) * (i − 1)⌉
8:         ST.stack[i] ← ST.stack[x]
9:         i ← x
10:    end while
11:    ST.stack[1] ← obj
12: end procedure
```

4.4 Implementation

Unlike the LRU stack, the KRR stack only shifts a small subset of objects on stack per stack update. To take advantage of that, we implemented the KRR stack as a simple array, where objects are ordered according to the stack order. When the object is referenced, we can find it in constant time using a hash table where a hash table

entry holds a pointer to the array location. An object's stack distance is simply its array index. On a stack update, first we identify all swap positions by using one of the algorithms described in Sections 4.3.1 and 4.3.2. Then, as shown in Figure 4.2 (b), we perform cyclic swapping on all marked positions. In our implementation, we adopted spatial sampling technique described in Section 2.4. By default, we use sampling rate of $R = 0.001$, but to ensure the accuracy of MRCs, a higher sampling rate is applied to workloads with relatively small working set sizes.

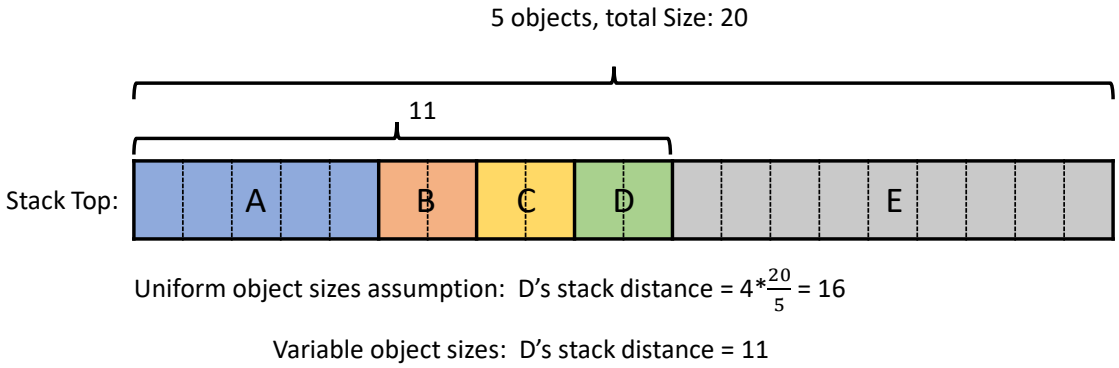


Figure 4.3: Byte-level Stack Distance Example

4.4.1 Handling Variable Object Sizes

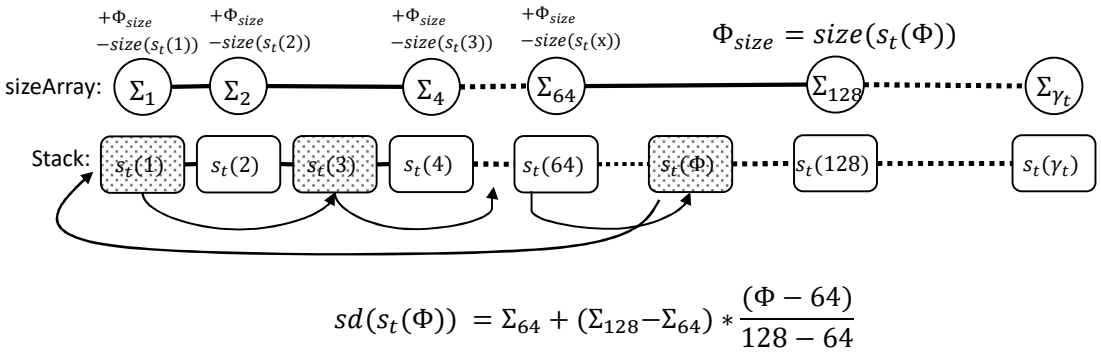


Figure 4.4: Variable Object Sizes Stack Update

The basic array implementation of the KRR stack implicitly assumes that all objects on the stack have identical sizes. Under such an assumption, the stack distance can be directly related to the object's array index. However, for workloads with diverse object size distribution [30], computing the object's stack distance base on its logical

location on the stack could be problematic. For example, in Figure 4.3, we see that, under uniform object sizes assumption, the estimated byte-level stack distance of object D (16) significantly differs from the actual byte-level stack distance (11). On each reference, to obtain the exact byte-level stack distance, one would need to sum up the size of all objects from the stack top to the referenced object, which indeed appears to be a very expensive task.

To collects byte-level stack distance efficiently, our solution is to add an additional array structure, *sizeArray*. Each entry of the *sizeArray* maintains a partial accumulation of stack size, specifically, the entry i of the *sizeArray* stores the total size of objects from stack top to stack position b^i, where b is the base parameter. Figure 4.4 illustrates the stack update process for a KRR stack with a base-2 *sizeArray*. Since the length of *sizeArray* is logarithmically bounded with respect to KRR stack length, the cost of maintain the *sizeArray* is at most $O(\log M)$, where M is the stack size. With aids from *sizeArray*, we can make better estimations on byte-level stack distance using Algorithm 3.

Algorithm 3 Byte-level KRR Stack Distance

```
1: procedure StackDistance(st, sizeArry, b, φ)
2:                                   ▷ st: Ordered Stack, implemented as an arrayList
3:                                   ▷ sizeArry: array of partial stack sizes
4:                                   ▷ b: sizeArray's base
5:                                   ▷ φ: stack position of referenced object
6:
7:     index ← log_b(φ)
8:     sdLow ← b^index
9:     if sdLow < φ then
10:        sdHigh ← b^(index+1)
11:        res ← (sizeArry[index + 1] − sizeArry[index]) ∗ (φ − sdLow)/(sdHigh − sdLow)
12:    else
13:        res ← 0
14:    end if
15:    return sizeArry[index] + res
16: end procedure
```

4.4.2 Available Artifact

The complete implementation of the KRR algorithm can be found here:

https://github.com/JYang1997/KRR-stack-algorithm

Chapter 5

Experimental Evaluation

This chapter evaluates KRR's accuracy, its time efficiency and space overhead.

5.1 Experiment Setup

The machine used for evaluation is configured with an Intel(R) Xeon(R) Gold 5118 2.30GHz processor with 30 MB shared LLC and 188 GB of memory, and the operating system is Fedora 31 with Linux kernel 5.6.15.

For comparison, we have implemented Mattson's LRU stack algorithm using a balanced search tree [17]. The conventional LRU stack can be implemented using a doubly-linked list which yields $O(M)$ per search and $O(1)$ per update. Using a balanced search tree results in $O(logM)$ for both search and update. This implementation can generate an accurate MRC for the true LRU policy. We also implemented SHARDS [24], which can output an approximated MRC for the true LRU policy.

To reveal the ground truth of the miss ratio of a K-LRU cache, we designed and implemented a cache simulator that adopts K-LRU replacement. A simulator can only generate one miss ratio for a given cache size with one pass of the input trace. To generate an MRC, we can run the simulator multiple times for different cache sizes and using interpolation for miss ratio prediction.

5.2 Workload Description

We use three different workloads for our evaluation:

MSR MSR Cambridge suite [1] is a collection of I/O traces from 13 different enterprise data center servers. We evaluate our model on all 13 traces, as well as the merged "master" MSR workload which is also used in Wire *et al* [24].

YCSB Yahoo Cloud Serving benchmark [2] is a well studied benchmark that provide a set of six different types of core workloads. In our evaluation, we use Workload C and E. Specifically, Workload C is a read-only workload follows Zipfian distribution, and Workload E is a scan dominant workload that uses Zipfian distribution to choose the first key in the range, and then uses uniform distribution to choose the number of objects to scan. For Workload E, we configure the max scan length to be the same as the number of distinct objects in the workload. We evaluate our model on both Workloads C and E, each with three different α values, 0.5, 0.99 and 1.5.

Twitter Twitter Cache traces [30] is a collection of one-week-long cache request traces from 54 Twitter's in-memory caching clusters. In our evaluation, we use 4 sub-traces, each with 100 million requests, from Twitter trace no. 26.0, 34.1, 45.0 and 52.7.

For Section 5.3, we convert every request to a standard "get/set" operation with uniform object size of 200 bytes.

For Section 5.4, which evaluates workloads with variable object sizes, we use both MSR and Twitter traces. For MSR traces, we convert every request to a standard "get/set" operation and use the block size from the first request to each object as the object's size. For Twitter traces, we use the original key, data size, and operation type.

5.3 MRC Accuracy

To measure the accuracy of the KRR model, we compare it with actual MRCs generated from directly simulating K-LRU cache under 40 different cache sizes that are

Table 5.1
Average MAE Under Different Sampling Size For MSR, YCSB and Twitter Traces.

	KRR						KRR+Spatial Sampling					
K	**1**	**2**	**4**	**8**	**16**	**32**	**1**	**2**	**4**	**8**	**16**	**32**
MSR	0.000079	0.00039	0.00052	0.00063	0.00067	0.00059	0.0015	0.0017	0.0017	0.0018	0.0019	0.0018
YCSB	0.000039	0.00079	0.0018	0.0029	0.0036	0.0036	0.0037	0.0039	0.0048	0.0058	0.0064	0.0063
Twitter	0.000016	0.00085	0.00057	0.000407	0.00029	0.00017	0.0017	0.0021	0.0018	0.0018	0.0017	0.0017

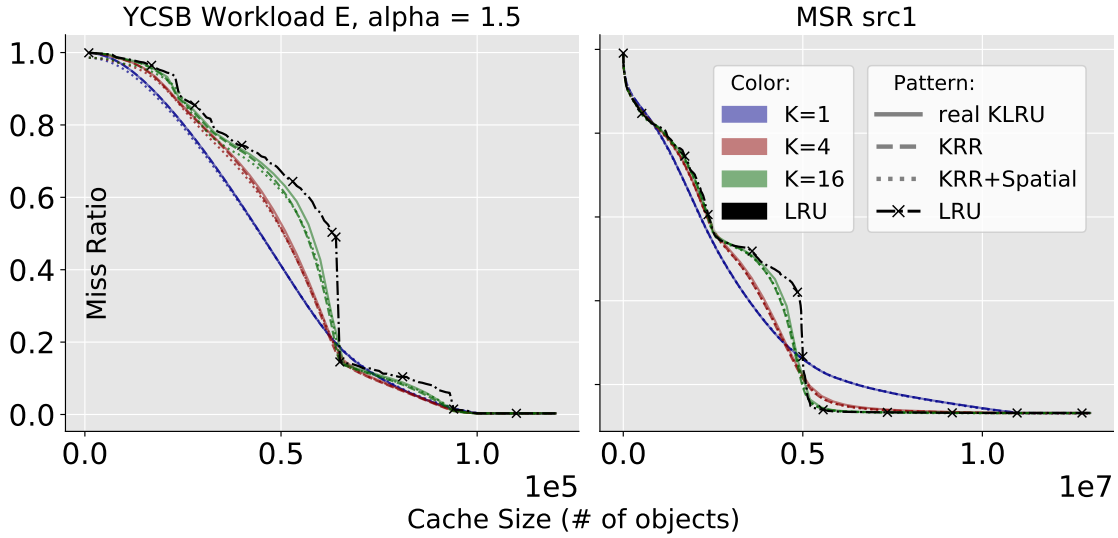


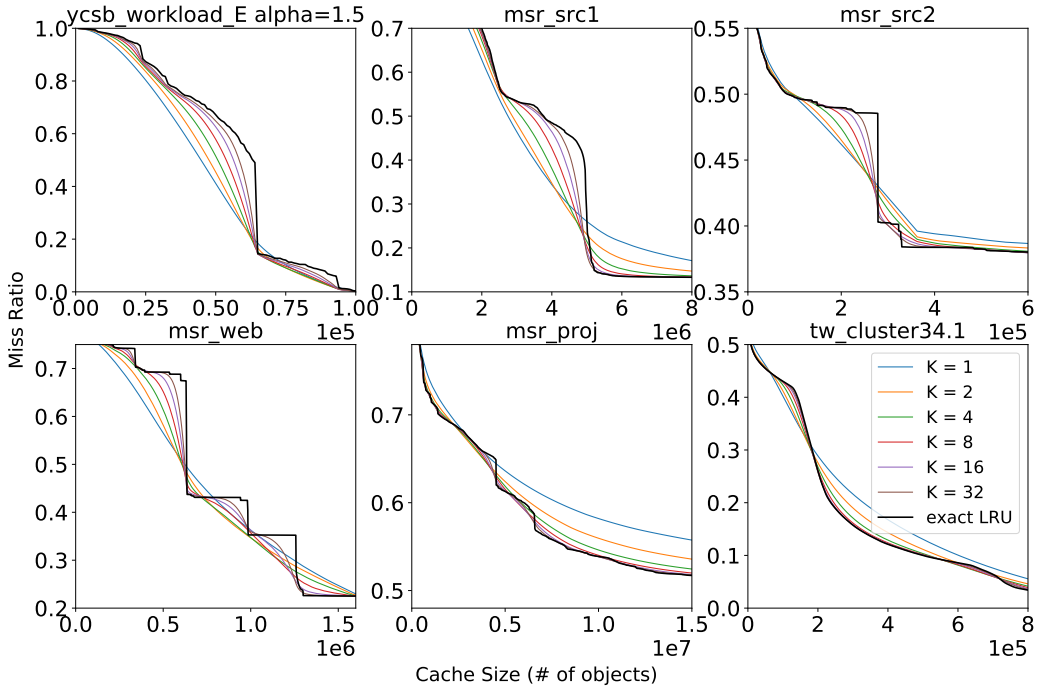
Figure 5.1: Actual vs. Predicted K-LRU MRCs. Three different colors represent K-LRU MRCs with K = 1, 4, 16, respectively. The actual and predicted K-LRU MRCs with and without spatial sampling are represented using different line types. The MRCs of exact LRU are plotted in black lines for comparison.

evenly distributed over the workload's working set size. Note that, since both stack update methods mentioned in Section 4.3 follow exactly the same swap probability, thus the accuracy of our KRR model does not depends on which method we use for stack update. For simplicity, all MRCs used in accuracy analysis are generated using the faster backward stack update. To quantify the accuracy of MRCs generated by the KRR model, we follow the error metric used in [24], the mean absolute error (MAE). The MAE between the actual and KRR MRCs is calculated as the mean of miss ratio differences across all simulated cache sizes. There are three sources of errors: (1) Simulation error. K-LRU and KRR are both probabilistic policies. There will always be a slight difference in miss ratio under different rounds of simulation. (2) Sampling error. Waldspurger *et al.* show that the spatial sampling error is inversely proportional to $\sqrt{n_s}$, where n_s is the amount of data sampled. Our default sampling rate is $R = 0.001$. To make the sampling error low, we apply a higher sampling rate to those workloads with a small working set size (less than 8M objects in our experiments) such that for all workloads we ensure there are at least 8K objects are

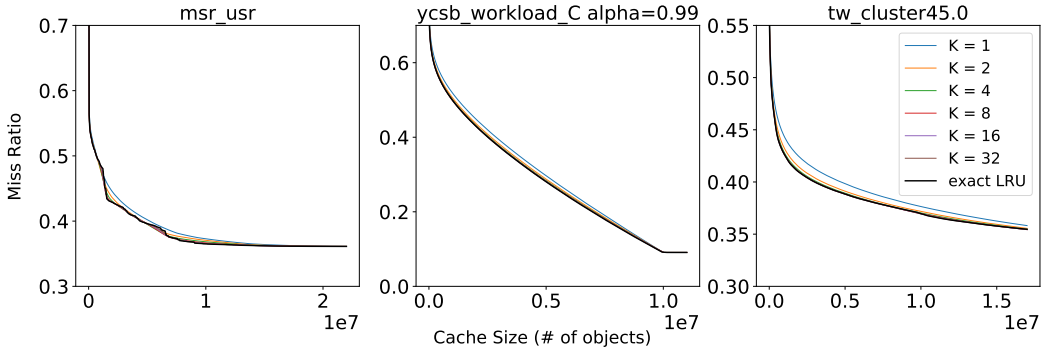
sampled. (3) Modeling error. As mentioned in Section 4.2, KRR and K-LRU are not statistically identical except when $K = 1$. In this section, our evaluation shows that for any arbitrary K, the MRC generated by KRR always closely approximates the K-LRU MRC.

We evaluate 13 MSR traces, 2 YCSB workloads each with three different α values (0.5, 0.99 and 1.5) and 4 Twitter sub-traces. Table 5.1 shows average MAE of all three types of workloads under different K values from 1 to 32 for both KRR and KRR with spatial sampling. Overall, we observe that the difference between actual and approximated MRCs is almost negligible. For KRR only, the average MAE across all traces with different sample sizes is 0.00099. KRR also works extremely well with spatial sampling, the average MAE across all traces is only 0.0026. The maximum MAE across all tested instances is around 0.01 from YCSB workload E with $\alpha = 0.99$ under KRR+Spatial sampling ($K = 2$). However, we notice that the MAE of the same trace under KRR without spatial sampling is negligibly 0.0003 which implies that a large portion error is contributed by spatial sampling not KRR itself. As an example, Figure 5.1 illustrates that MRCs generated by KRR are nearly identical to the actual MRCs for two representative traces, YCSB workload E with $\alpha = 1.5$ and MSR src1.

We observe that there are two different types of traces, A and B. Type A consists of the traces that have a notable difference in terms of MRC under different Ks. Type B consists of the traces that have nearly the same MRCs with respect to change in sampling size K. Figure 5.2 illustrates the MRCs of a few representative traces from both types A and B. Note that all traces in type A exhibit a significant gap between the LRU ($K = \infty$) MRC and the random replacement ($K = 1$) MRC. Thus, directly using true LRU's MRC to approximates the MRC of a K-LRU cache with a small K value ($K = 2, 4, 8$) can become very inaccurate. Existing fast MRC generation techniques such as AET, SHARDS and Counter Stacks are designed for LRU policy, use these techniques to approximate a K-LRU cache with a small K will not be reliable. From Corollary 1, we see that the number of swap positions increases as K increases. Thereby the stack update cost can be high with a large K such as $K \geq 32$. Fortunately, as illustrated by Figure 5.2, as K increases the K-LRU converges to LRU. For that reason, when approximating K-LRU with $K \geq 32$, directly applying an LRU MRC approximation technique such as SHARDS or AET would be more time efficient compared to KRR. For type B traces, the cache can yield similar miss ratios under different Ks. By choosing a smaller K, ($K = 1,\ 2$), we can effectively reduce the cost of sampling and eviction.



(a) Type A



(b) Type B

Figure 5.2: MRCs of Traces under K-LRU and LRU

5.4 Accuracy - Variable Object Sizes Workloads

To measure the effectiveness of our variable object size-aware KRR implementation, we evaluate our algorithms against variable object sizes MSR and Twitter traces (see Section 5.2). For convenience, we use *uni-KRR* and *var-KRR* to denote the uniform object size KRR and the variable size-aware KRR implementation described in Section 4.4, respectively. In Figure 5.3, we show MRCs from 8 different representative

traces (4 MSR and 4 Twitter). Each graph compares uniKRR and varKRR with the true MRC. We observe that the MRCs generated based on uniform size assumption (*uni-KRR*) does not always approximate the real MRCs well (shown in Figure 5.3(A)). In contrast, the *var-KRR* approximates the real MRCs with almost negligible errors. Table 5.2 summarized the MAE of MSR and Twitter traces under different K values from 1 to 32 for *var-KRR*. Overall, *var-KRR* achieves an MAE of 0.0008 (0.00143 with spatial sampling) for MSR traces and 0.00025 (0.00210 with spatial sampling) for Twitter traces.

Table 5.2
MAE Under Different Sampling Size for Variable Size MSR and Twitter Workloads

	Var-KRR		**Var-KRR+Spatial**	
K	**MSR**	**Twitter**	**MSR**	**Twitter**
1	0.00094	0.00023	0.00190	0.00201
2	0.00067	0.00045	0.00159	0.00213
4	0.00062	0.00034	0.00132	0.00176
8	0.00074	0.00018	0.00116	0.00165
16	0.00089	0.00013	0.00125	0.00238
32	0.00096	0.00014	0.00136	0.00268
Average	0.00080	0.00025	0.00143	0.00210

Table 5.3
Running Time Comparison for Processing One Million MSR src1 Requests

Stack Update Efficiency	
Methods	**Time (Sec)**
Simulation	26
Basic Stack	53606
Top Down Stack Update	97
Backward Stack Update	6.5
Top Down+Spatial	0.39
Backward+Spatial	0.07

5.5 Time Cost

To measure the efficiency of our stack update mechanisms, we compare both the top down and backward stack update methods (with/without Spatial sampling) with the

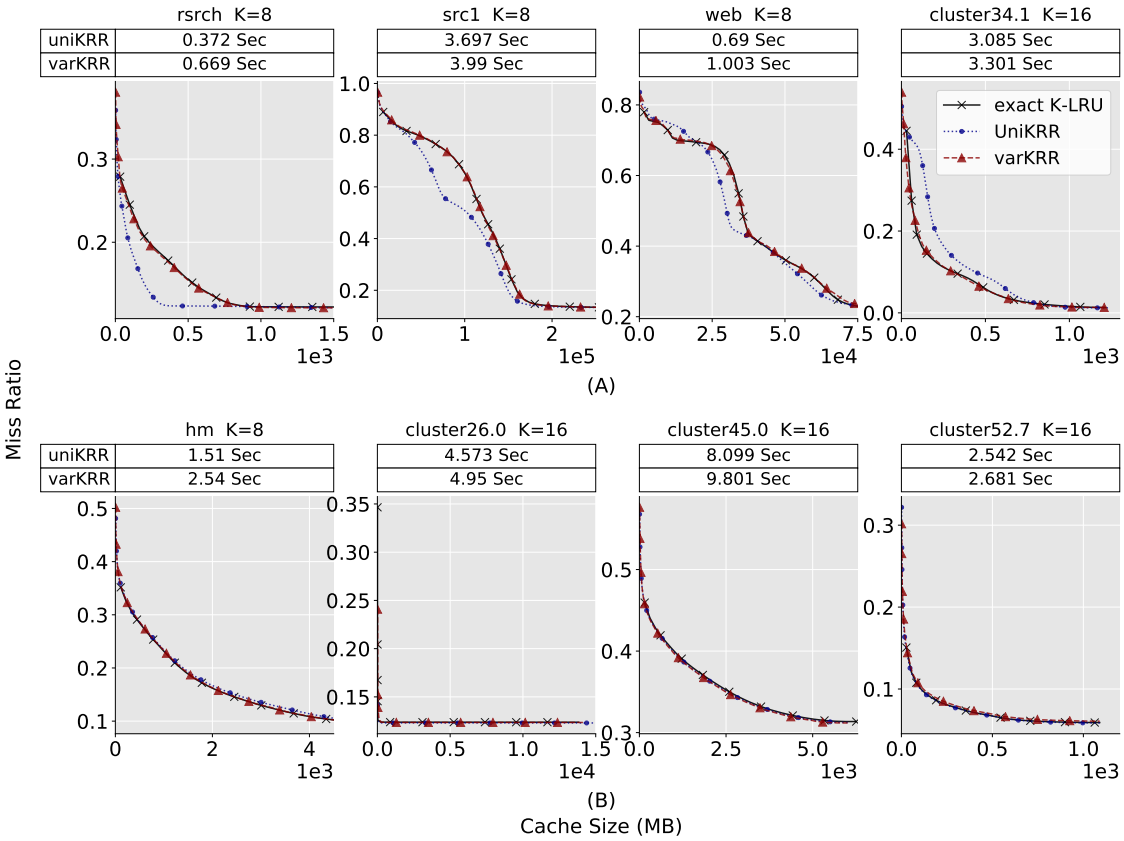


Figure 5.3: Accuracy and Time for Variable Size Aware KRR

naive linear stack update method and the simulation/interpolation-based approach. For interpolation, we simulate K-LRU under 25 different cache sizes evenly distributed across its working set size. For comparison, we use the first one million references from MSR src1 trace (mostly cold misses), and we use $K = 5$, the default sampling size for K-LRU in Redis. Table 5.3 is a summary of the results. We see that the top down stack update method shows x552 times improvement over the linear stack update approach, and the backward stack update method improves the run time overhead by x8247 times. When spatial sampling with $R = 0.01$[1] is applied, the running time is further improved by two more magnitudes. It is worth mentioning that even though generating MRC through interpolation may seem efficient in terms of running time, but there are several problems with using interpolation to generate MRCs for online applications. First, the accuracy and time overhead is directly associated with the number of cache sizes simulated. Second, for online applications, without the knowledge of the workload's working set size, it would be difficult to choose which cache sizes to simulate. Thus, an efficient one-pass algorithm is always preferred over

[1]We use $R = 0.01$ here, rather than $R = 0.001$ in other experiments, to keep the sampling error low by ensuring at least $8K$ objects are sampled over the one million requests.

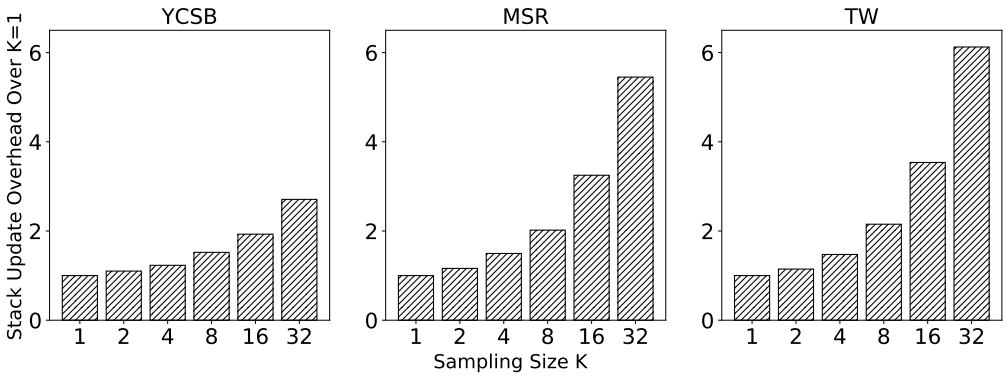


Figure 5.4: Normalized Average Stack Update Overhead Against K=1

interpolation. Next, we use the merged "master" MSR trace to compare the running

Table 5.4
Master Trace Comparison

Merged-MSR Trace, Spatial Sampling Rate = 0.001			
Method	Top Down+Spatial	Backward+Spatial	SHARDS
Times (sec)	39.1	22.4	19.7

time of KRR+Spatial sampling with existing LRU MRC approximation technique, SHARDS. Table 5.4 contains running time for both versions of KRR and SHARDS. The running time of KRR shown in Table 5.4 is the average across different Ks $(1, 2, 4, 8, 16, 32)$. The average running time for KRR with backward stack update and SHARDS are very close in our test. The top down stack update method on average is about two times slower than SHARDS for the master trace.

Corollary 1 shows that the number of expected swap positions is proportional to sampling size K. As the number of swap positions increases, the expected cost of stack update also increases proportionally. Figure 5.4 shows the normalized average running time overhead against K=1 across all traces. The time overhead for $K \leq 16$ is generally no more than 4 times greater than that of $K = 1$ in our tests. As mentioned in Section 5.3, as K increases, K-LRU converges to true LRU. Therefore, when approximate MRC for K-LRU with a large K ($K \geq 32$), conventional LRU MRC approximation techniques like SHARDS are recommended.

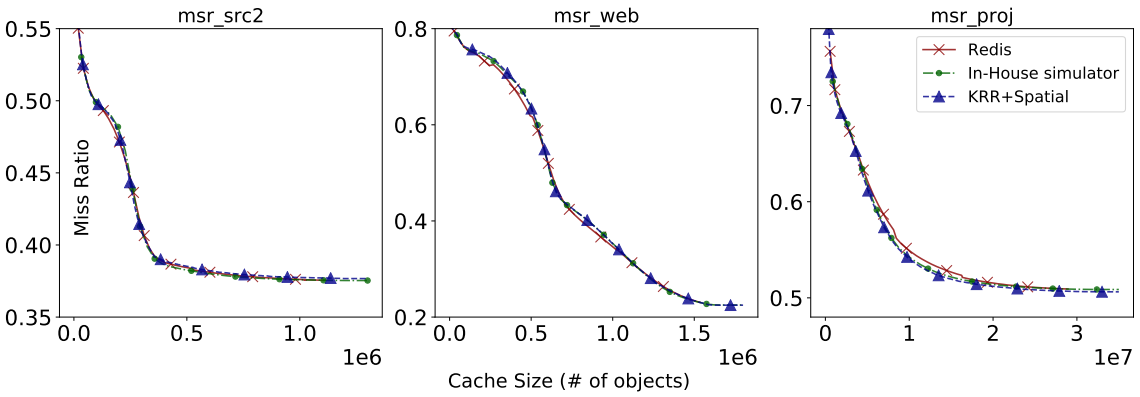


Figure 5.5: Validating the KRR with Redis. The Redis MRCs are generated by running Redis instances with 50 different memory sizes.

5.6 Space Cost

The KRR stack is implemented as a simple array with a hash table where an entry of the hash table holds a pointer to an object location in the array. Then the total space overhead of the KRR stack is proportional to the total number of objects stored on the KRR stack. In our implementation, each object consumes 68 bytes including hash table and other auxiliary entries. For *var-KRR*, a 4 bytes field is needed to store the size of each object, the additional *sizeArray* consumes negligible space in comparison to the stack. After incorporating spatial sampling, the overall space overhead is further reduced by sampling rate R. Thus the estimated percentage of space overhead is *72 bytes * R / average object size*. For instance, assuming R = *0.001*, and the average size of objects is 200 bytes[2], then the space overhead for processing a workload with 100 million distinct objects is just 0.036% of the working set size.

5.7 Validation of KRR on Redis

In this section, we validate KRR against Redis, the real world in-memory key-value store. We compare the actual Redis MRCs with the KRR model, as well as the MRCs from our K-LRU cache simulator. The Redis MRCs are generated by running Redis instances with 50 different memory sizes. In general, as shown in Figure 5.5, the KRR

[2]Many real in-memory cache workloads have much higher average KV size [30]

approximation is highly accurate when compared to Redis' MRCs. One might notice that there is a slight deviation between MRCs from the K-LRU cache simulator and Redis. This is because Redis uses a different sampling mechanism to sample K objects from the cache. For the purpose of efficiency during sampling, the mechanism does not guarantee that sampled objects follow a good random distribution. [3]

To evaluate the overhead of KRR in Redis, we set the object size of all objects, in the traces shown in Figure 5.5, to 200 bytes, and configure Redis memory size to approximately 50% of the working set size of each trace. The average of three trials of experiment on each trace reveals that only 0.08%, 0.11%, and 0.09% of total execution time are consumed by KRR, respectively. The space overhead is also extremely small. For all three traces, the space consumed by the KRR stack never exceeds 1MB.

[3]Redis also provides an alternative sampling mechanism, "*dictGetRandomKey()*", which is less efficient than the default method but guarantees good random sampling. This sampling mechanism yields nearly identical miss ratio curves to the ones by our K-LRU cache simulator.

Chapter 6

Related Work

The Least Recently Used replacement (LRU) strategy is the most well-known replacement strategy due to its simplicity and effectiveness. Many advanced recency-based replacement strategies are more or less extensions of LRU replacement. Thus, many prior works on cache modeling often assume the cache is configured with LRU replacement. This chapter highlights some research works on cache modeling that are not mentioned in previous chapters. Specifically, in Section 6.1, we describe few representative works on LRU miss ratio curve generation. Then, in Section 6.2, we describe three works that aim to efficiently model cache beyond LRU replacement.

6.1 LRU MRC Techniques

The baseline technique by Mattson *et al.* [14] simulates a linear LRU stack to track stack distance, which can be used to construct the MRC for an LRU cache through a single pass of the trace. The original stack algorithm tracks exact stack distance for every access, which generates exact MRC for the LRU cache but comes with the cost of extremely large space and time overheads. However, in practice, the optimization decision made based on MRC are often in much coarser granularity [10, 19]. Many later works attempted to reduce the overhead of stack processing by using more compressed stack representation.

Scale Tree [32] is a modified version of Olken's stack [17]. Instead of using each node to store exactly one reference, the scale tree stores a time range of references in

one node. Compressing multiple references into one node is essentially a trade-off between error and space/time overheads. The scale tree approximates stack distance with a small bounded error in which only takes $O(NLog(Log(M)))$ time and $O(LogM)$ space.

MIMIR [22] divides the LRU stack into B variable size buckets, in which the elements can be in any order within a bucket. The sequence of buckets forms a coarser-grained LRU stack. To obtain the stack distance of a reference in bucket B_i, we simply sum up the size of all buckets before from B_0 to B_{i-1} which gives a rough estimate of its stack distance. This method takes $O(B)$ time and $O(M)$ space. They demonstrated that with $B = 128$, MIMIR can generate very accurate MRCs.

Counter Stacks [26] replaces the original LRU stack with a set of cardinality counters. Each cardinality counter stores the total number of unique references observed since the counter initialized. The basic idea for Counter Stacks is that the LRU stack distance is just counting the number of unique references between re-references. Thus, the LRU stack processing can be considered as a stack of cardinality counters, one for each request. To make it practical for online use, Counter Stacks employs multiple compression techniques includes downsampling and pruning its data matrix as well as replaces the bloom filter-based counter with a low overhead probabilistic cardinality counter. The compressed Counter Stacks only requires $O(NLogM)$ time and $O(LogM)$ space to generate accurate MRCs with bounded error.

More recent advancement in LRU MRC generation leverages the metric called reuse time. reuse time is defined as the total number of references between two references to the same object. These techniques do not explicitly maintain any representation of stack when processing the workload, instead, they collect the reuse time distribution of the workload through sampling which can be done in just linear time with a small fraction of space overhead.

Statstack [9] converts the reuse time distribution to an expected stack distance distribution. For every reference with reuse time r, or equivalently there is r number of references in between the re-reference, they approximate the expected stack distance as the expected number of references out of the r references that have forward reuse time greater than r.

In **HOTL** [27], Xiang *et al.* shows that miss ratio of an LRU cache with capacity c can be approximated by finite difference of average footprint at c, which is equivalent to the fraction of reuse time longer than the footprint window.

A recent work, **AET** [12], presents a kinetic model for LRU cache eviction process. This model use reuse time distribution to computes the object's movement probability (or equivalently its instantaneous velocity) at an LRU stack position. For a reference, given its reuse time, the model computes the approximated stack distance by integrating the reference's moving speed over reuse time.

In terms of correctness, these reuse time-based models do not always produce accurate MRC. Their correctness often depends on how close the workload follows their assumptions. For example, one common hidden assumption among these models is that the reference's reuse time is independent from each other, and the reuse time distribution is static over time, which is not necessarily true for many workloads. Nonetheless, much empirical evidence shows that these reuse time-based models work very well in practice, large errors only occur in a small circumstance where the correctness conditions are awfully violated.

6.2 Generic MRC Techniques

Min-Tree Algorithm. Bilardi *et al.*, proposed a new representation for stack processing called Min-Tree and introduced a class of replacement policies called NSP [7]. A replacement policy belongs to the class NSP if, under such policy, the priority of an item only changes upon access to that item. Replacement policies such as OPT, LFU, LRU, and MRU belong to the class NSP. Under Min-Tree representation the cost of processing certain NSP stacks can be significantly reduced. Bilardi *et al.* show that the time cost of the Min-Tree algorithm is depending on the expected number of swap positions per stack update. For a stack with D expected swap positions per update, Min-Tree is expected to complete the stack update in $O(D * Log\phi)$ time. When compared to the original linear stack, Min-Tree representation shows significant performance improvement over policies that have a small expected number of swap positions, such policies include LFU, MRU, and OPT.

Miniature Cache Simulation. The key insight behind many efficient single-pass MRC construction algorithms is that many replacement policies satisfy inclusion property (2.2). Unfortunately, for many non-stack policies, such as ARC [15], there is no known single-pass solution. Thus, the only option for constructing MRC for these policies is to emulate each different cache size. Waldspurger *et al.* proposed the miniature cache simulation, which emulates a given cache size using a scaled-down miniature cache over a spatially-hashed sample of requests [23]. Like SHARDS (Section 2.4), the miniature simulation can achieve extremely high accuracy even under

a sampling rate of $R = 0.001$. This allows generating MRC by emulating multiple cache sizes under relatively small space/time overheads.

LLC Modeling. The accesses to LLC are typically filtered by upper-level private caches, so the input stream of LLC is typically stripped from temporal correlation. Motivated by such property of LLC, Beckmann *et al.* developed a generic cache model that predicts the performance of age-based replacement policies on modern LLCs [4]. The model established a relationship between reuse time distribution, hit and evict distribution, then it takes reuse time distribution of the workload as the input solves hit and evict distribution through a fixed point iteration. Although the original work does not provide rigorous convergence criteria, their empirical results show the model makes a very accurate prediction when the model assumption holds.

Chapter 7

Conclusion

Random sampling-based cache replacement policies such as K-LRU become more attractive recently due to their small metadata and data structure maintenance overhead, and acceptable miss ratio. However, modeling these policies remains a challenging problem. This paper presents KRR, a probabilistic stack algorithm which enables MRC construction for variable block size K-LRU cache in one pass of the trace. Moreover, we propose two fast stack update schemes to further reduce the algorithm's cost. Incorporating spatial sampling, we show that KRR can construct an accurate MRC with very low space and time overhead. In our future work, we will investigate other random-sampling policies which use other metrics, such as access frequency and object expiration time, as priority functions.

References

[1] [n.d.]. MSR Cambridge Traces. `http://iotta.snia.org/traces/388`. Accessed: 2020-03-15.

[2] [n.d.]. Yahoo! Cloud Serving Benchmark (YCSB). `https://github.com/brianfrankcooper/YCSB`. Accessed: 2020-03-15.

[3] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. `https://www.usenix.org/conference/nsdi18/presentation/beckmann`

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

[5] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. `https://www.usenix.org/conference/osdi20/presentation/berg`

[6] Erik Berg and Erik Hagersten. 2004. StatCache: a probabilistic approach to efficient and accurate data locality analysis. In *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 20–27.

[7] Gianfranco Bilardi, Kattamuri Ekanadham, and Pratap Pattnaik. 2011. Efficient Stack Distance Computation for Priority Replacement Policies. In *Proceedings of the 8th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '11)*. Association for Computing Machinery, New York, NY, USA, Article 2, 10 pages. `https://doi.org/10.1145/2016604.2016607`

[8] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 499–511. https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein

[9] David Eklov and Erik Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*. IEEE, 55–65.

[10] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX ATC*.

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

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

[13] Redis Labs. 2020. *redis*. Retrieved September 10, 2020 from https://redis.io

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

[15] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *2nd USENIX Conference on File and Storage Technologies (FAST 03)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache

[16] memcached. 2020. *memcached*. Retrieved September 10, 2020 from https://memcached.org

[17] Frank Olken. 1981. *Efficient methods for calculating the success function of fixed-space replacement policies*. Technical Report. Lawrence Berkeley Lab., CA (USA).

[18] Cheng Pan, Xiameng Hu, Lan Zhou, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. PACE: Penalty Aware Cache Modeling with Enhanced AET. In *Proceedings of the 9th Asia-Pacific Workshop on Systems* (Jeju Island, Republic of Korea) *(APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 19, 8 pages. https://doi.org/10.1145/3265723.3265736

[19] Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2019. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 193–205. https://doi.org/10.1145/3357223.3362729

[20] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, Washington, DC, USA, 423–432. https://doi.org/10.1109/MICRO.2006.49

[21] Liana V. Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. 2021. Learning Cache Replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. USENIX Association, 341–354. https://www.usenix.org/conference/fast21/presentation/rodriguez

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

[23] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *Proceedings of USENIX ATC*. 487–498.

[24] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, 95–110.

[25] Yuchen Wang, Junyao Yang, and Zhenlin Wang. 2020. Dynamically Configuring LRU Replacement Policy in Redis. In *The International Symposium on Memory Systems* (Washington, DC, USA) *(MEMSYS 2020)*. Association for Computing Machinery, New York, NY, USA, 272–280. https://doi.org/10.1145/3422575.3422799

[26] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, Andrew Warfield, and Coho Data. 2014. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*. USENIX Association, 335–349.

[27] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: a higher order theory of locality. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 343–356.

[28] Yaocheng Xiang, Xiaolin Wang, Zihui Huang, Zeyu Wang, Yingwei Luo, and Zhenlin Wang. 2018. DCAPS: Dynamic Cache Allocation with Partial Sharing. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 13, 15 pages. https://doi.org/10.1145/3190508.3190511

[29] Junyao Yang, Yuchen Wang, and Zhenlin Wang. 2021. Efficient Modeling of Random Sampling-Based LRU. In *50th International Conference on Parallel Processing* (Lemont, IL, USA) *(ICPP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 32, 11 pages. https://doi.org/10.1145/3472456.3472514

[30] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

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

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

[33] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*. USENIX Association, USA, 91–104.