Multi-Tenant In-memory Key-Value Cache Partitioning Using Efficient Random Sampling-Based LRU Model

Yuchen Wang, Junyao Yang, and Zhenlin Wang

Abstract—In-memory key-value caches are widely used as a performance-critical layer in web applications, disk-based storage, and distributed systems. The Least Recently Used (LRU) replacement policy has become the *de facto* standard in those systems since it exploits workload locality well. However, the LRU implementation can be costly due to the rigid data structure in maintaining object priority, as well as the locks for object order updating. Redis as one of the most effective and prevalent deployed commercial systems adopts an approximated LRU policy, where the least recently used item from a small, randomly sampled set of items is chosen to evict. This random sampling-based policy is lightweight and shows its flexibility. We observe that there can exist a significant miss ratio gap between exact LRU and random sampling-based LRU under different sampling size Ks. Therefore existing LRU miss ratio curve (MRC) construction techniques cannot be directly applied without loss of accuracy. In this paper, we introduce a new probabilistic stack algorithm named *KRR* to accurately model random sampling based-LRU, and extend it to handle both fixed and variable objects in key-value caches. We present an efficient stack update algorithm that reduces the expected running time of KRR significantly. To improve the performance of the in-memory multi-tenant key-value cache that utilizes random sampling-based replacement, we propose kRedis, a reference locality- and latency-aware memory partitioning scheme. kRedis guides the memory allocation among the tenants and dynamically customizes K to better exploit the locality of each individual tenant. Evaluation results over diverse workloads show that our model generates accurate miss ratio curves for both fixed and variable object size workloads, and enables practical, low-overhead online MRC prediction. Equipped with KRR, kRedis delivers up to a 50.2% average access latency reduction, and up to a 262.8% throughput improvement compared to Redis. Furthermore, by comparing with pRedis, a state-of-the-art design of memory allocation in Redis, kRedis shows up to 24.8% and 61.8% improvements in average access latency and throughput, respectively.

Index Terms—Application servers, multi-tenant, LRU, modeling methodologies, memory allocation.

1 Introduction

To reduce the latency of accessing back-end servers, today's web services pervasively adopt in-memory key-value caches in the front end which cache frequently accessed objects. For large-scale web services, key-value caches like Redis and Memcached [1], [2] are crucial to ensure low-latency service when serving enormous workloads. Compared to a dedicated cache that serves a single application usage, a multi-tenant cache allows multiple applications to share a single cache instance, where the available memory is partitioned for each tenant to meet their caching requirements. Due to the limited size of memory, an in-memory key-value cache needs to be configured with a fixed amount of memory, thus maximizing memory utilization of the shared memory is a key to delivering the best system performance.

In a long history, the miss ratio curve (MRC) has been a useful tool for cache memory management [3], [4], [5], [6], [7], [8], [9], which is a function mapping from cache sizes to miss ratios. Given the MRC of a workload, one can immediately know the miss ratio for any cache allocation. Since accesses in real-world commercial key-value caches show high data locality [5], [10], LRU becomes a good choice as it can exploit the locality well. As of today, most studies on efficient MRC construction are focused on the LRU cache [3], [6], [11], [12], [13]. However, exact LRU implementation can be costly. In software caches, prioritizing items according to their last-access-time usually relies on linked structures to book-keep their orderings [14], and item evictions require list operations including pointer updates. All of these introduce space overhead and computation overhead. In addition, each time when an item is accessed, the LRU list must be locked to facilitate the update of corresponding LRU priority, resulting in extra performance degradation [15]. Memcached, another popular key-value cache, only maintains the LRU structure at the slab class level [2].

To avoid using expensive ordered-data structures and to improve performance, many existing schemes have adopted the idea of random sampling: on eviction, the cache randomly selects a small number of items and then evicts the 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 [14]. The commercial in-memory cache, Redis, applies an approximated LRU policy [19] that only needs to keep track of the access time of each object. On an eviction, the candidate with the oldest access time is selected from a pool consisting of randomly sampled keys. Experiments have demon-

- *Y. Wang, J. Yang, and Z. Wang are with the Department of Computer Science, Michigan Technological University, Houghton, MI, 49931. E-mail: {yuchenwa, junyaoy, zlwang}@mtu.edu*
- *Y. Wang and J. Yang contributed equally to this work.*
- *Corresponding author: Zhenlin Wang.*

strated that, with a relatively small sampling size, random sampling-based LRU closely approximates exact LRU. For simplicity, we use K-LRU to denote a random sampling-based LRU policy, where K represents the sampling size.

Years of research have improved the asymptotic complexity of modeling exact LRU MRC. For a workload of N references to M distinct objects, the baseline Mattson's stack distance algorithm takes $O(NM)$ time. More recent designs, such as Footprint [12], Statstack [13], and AET [21], improve it to $O(N)$. However, those algorithms may lose accuracy in K-LRU (see Section 2.2). Additionally, the recent research on cache-sharing models that guide memory allocation among the tenants, including LAMA [5], mPart [22], pRedis [24], and Memshare [25], are all based on the exact LRU policy. To the best of our knowledge, the memory management for the multi-tenant key-value cache that employs K-LRU still needs to be addressed. This paper develops an efficient algorithm to model random sampling-based LRU and construct the K-LRU MRC accurately. Using such a model, we propose a multi-tenant cache memory partitioning scheme to improve cache performance. This paper makes the following contributions:

- To accurately model the behavior of the 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 [41], a stack algorithm statistically equivalent to the random replacement policy.
- We propose an efficient backward stack update mechanism that reduces KRR's expected running time from $O(NM)$ to $O(NlogM)$. Together with the spatial sampling technique [3], we further reduce the time overhead to a tiny magnitude which makes it practical for constructing a K-LRU MRC online. We extend KRR by applying a new byte-level stack distance estimation algorithm to accurately handle variable object sizes.
- We propose *kRedis*, a multi-tenant memory partition system, which is guided by merged K-LRU MRCs and is aware of tenant miss latency. Inspired by our previous work, DLRU, which explores the configurable random sampling size K to improve the single-tenant in-memory cache performance [27], this paper presents a new memory arbitration scheme dynamically configures the random sampling size K for each individual tenant to adapt to access pattern changes, exploring the possible miss ratio gap between various K options. We adopt a new multi-dictionary design to increase the efficiency of random sampling for the individual tenants.
- Using MSR and Twitter workloads [29], [32], we show that KRR yields a highly accurate MRC for K-LRU cache with low space and time overhead. And the performance evaluation shows that kRedis attains up to a 50.2% lower average access latency, and up to a 262.8% higher hit throughput than Redis. When compared to pRedis, a state-of-the-art design of memory allocation, kRedis yields up to 24.8% and 61.8% improvements in access latency and throughput, respectively.

The rest of the paper is organized as follows. We review MRC construction techniques and the opportunities in designing latency- and locality-aware caching in Section 2. We present the K-LRU MRC model and its extension to handle variable object sizes in Section 3. We develop the merged-MRC guided and latency-aware memory partitioning system on top of a multi-dictionary design of Redis in Section 4, and present its implementation in Section 5. Evaluation results are shown in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2 Background

We first briefly describe the classic single-pass MRC construction algorithm, namely Mattson's generic stack algorithm. Next, we address the motivation for exploring the miss ratio gap of K-LRU for different Ks, and the DLRU model that dynamically explores it in a single-tenant key-value cache. Then, we describe the challenges of memory partitioning for multi-tenant cache environments and the motivation for handling variable-sized objects in the Redis cache. Lastly, we use an example to address the motivation of combining the miss latency into the memory allocation strategy and describe the spatial sampling technique adopted from SHARDS [3].

2.1 Stack Algorithm

Mattson's Stack Algorithm is a generalized algorithm that models a general class of replacement policies. 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 the stack algorithm makes it possible to generate an MRC in just one pass of the trace.

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.

Under the general stack model, all previously referenced objects have an associated priority. Depending on the replacement policy, an object's priority can change over time. At any given time, all referenced objects' 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 the following constraints:

$$
\begin{aligned}
&S_t(1) = x_t \\
&S_t(i) = maxPriority\left(y_t(i-1), S_{t-1}(i)\right) && \text{for } 2 \le i < \phi \\
&S_t(\phi) = y_t(\phi - 1) \\
&S_t(j) = S_{t-1}(j) && \text{for } \phi < j \le \gamma_{t-1}
\end{aligned}
$$

where:

x_t : object referenced at time t

$S_t(i)$: object at i_{th} stack position at time t.

$y_t(i)$: 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, if x_t never referenced, $\phi = \gamma_t$

The $maxPriority()$ function in the stack maintenance procedure above is a function comparing the 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 the cache of size i. For simplicity, one can think that the only difference among stack algorithms is their $maxPriority()$ function. The stack update process 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) 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 pushed 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 the expected linear time with respect to the stack size. The running time of Mattson's LRU stack algorithm is thus $O(NM)$.

2.2 Random Sampling Replacement and K-LRU Policy

In Redis' random sampling-based LRU, each time when an eviction is needed, K keys are randomly sampled from all keys in the memory and added to the eviction pool. All keys in the eviction pool are sorted by their last access time and the one with the oldest time is evicted. The default setting of Redis is $K = 5$. Each time, 5 randomly sampled keys are added to the pool for the eviction decision. The number of sampled keys, K, is configurable but is fixed across Redis execution for the current design. Redis [19] shows that $K = 5$ is good for picking the approximate LRU keys while saving memory and CPU usage. And it recommends raising K to 10 for a closer approximation of exact LRU.

We run a collection of real-world enterprise server traces from Microsoft Research Cambridge [29] on Redis to plot the MRCs. Figure 1 shows the MRCs of six different sampling size Ks for trace `web` where cache size is represented as the number of objects. We can clearly observe that the sampling size K could impose large impacts on miss ratios. When the cache size is less than 5 * 1e5, the miss ratio of K-LRU with $K = 1$ is almost always lower than other settings of K, which means that the random replacement policy can perform better. As pointed out by Jaleel et al. [35], LRU is not able to explore well the reuse intervals (reuse distances) that are larger than the cache size. (Note that the reuse distance between access and its next reuse is the number of distinct accesses in between.) For trace `web`, random replacement performs better when the cache size is tight. However, when the cache size grows large, the MRC apparently indicates that other options of K are better choices if we only consider the impact of the miss ratio. The maximum absolute difference of miss ratio under various K settings at a fixed cache size could be more than 10%.

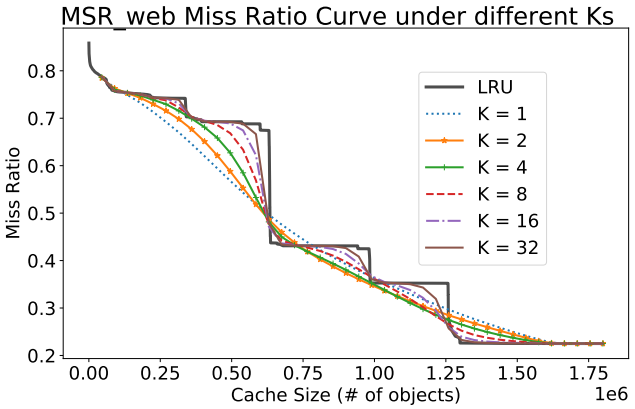


Fig. 1. MSR web K-LRU MRCs.

We observe that the MRCs of K-LRU with $K = 16$ can closely approximate those of the real LRU. A larger K normally indicates a closer behavior of eviction to the exact LRU policy, since the more keys are sampled to add into the eviction pool, the more likely the evicted object is near the least-recently-used side of the exact LRU list.

2.3 Dynamic LRU (DLRU)

Observing the potential miss ratio difference in K-LRU eviction policy with various K options, our previous work introduces DLRU [27], a method that dynamically adjusts K to improve the performance of the single-tenant key-value cache. DLRU utilizes a scaled-down cache simulator, miniature cache [28], which monitors the miss ratios of various Ks with low overhead. Each K is assigned a specific miniature cache. A penalty cost model is employed to select a K that optimizes the overall miss latency and is applied to reconfigure Redis in real-time. Experiments show that DLRU is capable of adapting to workload access pattern changes and consistently surpasses a fixed-K system across diverse traces.

2.4 Memory Partitioning

In real-world cache deployments, one cache instance usually serves multiple tenants, forming a multi-tenant environment. There are two basic memory partitioning strategies, equal partitioning, and free competition. In equal partitioning, each of the n tenants running simultaneously takes *1/n* of the total memory. This strategy seems to be fair to all tenants. However, due to access pattern change or trace locality difference, cache performance might deteriorate by offering some tenants more memory than needed, while others suffer from memory shortage. Free competition, on the other hand, is a first-come-first-serve policy, all tenants are competing for shared memory. The memory usage of tenants is decided according to various factors, including access rate, locality, object size, miss latency, etc. This is the default strategy that Redis employs. A tenant's throughput may be significantly affected by some noisy neighbors. To

maximize the memory utilization of high-throughput multi-tenant storage systems, recent studies consider better memory partitioning schemes for applications based on online MRC construction techniques. For example, LAMA [36] adopts footprint theory [12], [37] while Dynacache [38] applies bucket algorithm [39], and both mPart [22] and pRedis [24] utilize AET [21] for online MRC construction. A variety of optimization strategies have been applied in memory partition including minimizing the overall miss ratio [22], [36], [38] or the average response time [24], [36]. However, those MRC techniques are designed for the exact LRU policy, which is not accurate for workloads with miss ratio sensitive to sampling size K as demonstrated in Sections 2.2 and 6.3.

2.5 Variable Object Size

One of the key features distinguishing in-memory key-value caches from other caching types is the variable object size distribution. Recent studies show that the size of objects in the in-memory cache can be very diverse. For example, Facebook's caching systems can deal with object sizes that span over seven orders of magnitude [43]. Also, the size distribution of workloads is usually not static over time, it can display both periodic shifts and sudden changes, as observed in systems like Twemcache [32], affecting both miss ratio and throughput. For slab-based caching systems such as Memcached, the non-static object size distribution can cause slab calcification [5], [32]. For Redis employing heap memory allocators such as Jemalloc, the dynamic object size can pose challenges to memory fragmentation management. Also, object size affects the time to fetch objects from the DB or over the network, the larger the size, the longer the time it takes for the request. Moreover, Handling variable-size objects in Redis holds particular importance given the complexity and richness of its internal data structures, which markedly differ from those in systems like Memcached. Redis supports a diverse set of data structures, including strings, linked lists, arrays, sets, hash tables, and ordered lists, amongst others. Due to this diversity, the metadata size of the data structure varies with data type and size. For instance, the String Object in Redis, the simplest type, has three different encoding methods - int, embstr, and raw - each with its own memory allocation strategy depending on the length of the string. Pan *et al.* [44] 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. We also discuss this observation in Section 6.3.

2.6 Miss Latency

Research has shown that the overall performance of a cache system is highly determined by its miss ratio – even a slight reduction of it could introduce a significant improvement in performance [22], [25]. To further estimate the performance impact of misses, we need to know the related latency. In this paper, the miss latency is defined as the time interval from the miss of a GET operation in the key-value cache to the completion of a SET operation with the same key sent by the client. Previous work [24] shows the distribution of miss latency in real-world cloud computing can be widespread in the range between 10's to 10000's of microseconds depending on the locations of the back-end servers.

We use a multi-tenant example to show how miss latency impacts cache performance. Assume that one networked cache instance is serving two co-running tenants. The hit time of each object is 40 μs, while the miss latency of each tenant is 2000 μs and 12000 μs, respectively. We also assume that all tenants use equal-sized memory, they send requests at the same rate, and the miss ratio of all tenants is 0.5. The average access latency can be expressed as:

$(0.5\times40+0.5\times2000)+(0.5\times40+0.5\times12000)=7040$ μs.

According to MRC profiling, if we can actively, for instance, decrease the memory usage of low miss latency tenant while increasing that of high miss latency tenant, such that the miss ratios have been changed to 0.8 and 0.1, respectively. Then the average access latency becomes:

$(0.2\times40+0.8\times2000)+(0.9\times40+0.1\times12000)=2844$ μs.

The average access latency difference between the two memory partitioning cases cannot be ignored. Memory partitioning in a multi-tenant key-value cache must consider both miss ratios and miss latencies.

Though average access latency is an essential metric for in-memory key-value cache, other metrics such as throughput, back-end database load, tenant fairness, etc. need to be considered for use-case-specific needs. A memory partitioning scheme should be capable of adjusting its optimization target according to variable performance requirements.

2.7 Spatial Sampling

For any stack algorithm, an MRC can be calculated from the generated SDH. 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 a stack algorithm online. In order to make it suitable for online usage, the uniformly random spatial sampling described in SHARDS [3] has become a widely adopted technique. Instead of feeding entire reference streams to the stack model, the 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. The effective sampling rate is $R = T/P$. As shown by Waldspurger *et al.*, for the majority of workloads tested, the sampled subset has very high statistical similarity compared to the original workload, even with R = 0.001. The spatial sampling technique can thus significantly reduce the number of tracked references for online MRC prediction.

3 K-LRU Replacement Model

As demonstrated by Figure 1, a 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[1] only model stack distance distribution for caches under the exact LRU policy.

1. The SHARDS here specifically refers to the spatially scaled-down version of the LRU balanced tree, not the spatial sampling method.

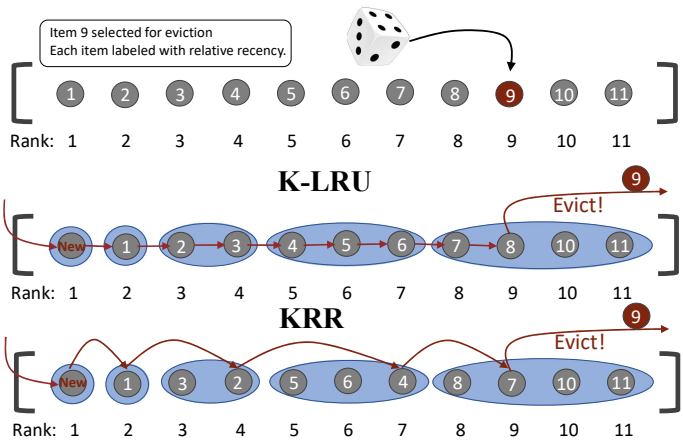


Fig. 2. Eviction comparison between K-LRU and KRR.
The red edge represents the movement of objects' ranks. The blue oval groups a coarse-grained ordering of objects' recency.

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.

3.1 KRR Stack Algorithm

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 removed from the 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 the highest ranking, that is, the object most unlikely to be evicted. We define $\rho_{t,C}(r) : r \rightarrow d$ as the mapping function that maps the object's recency r to the object's relative priority ranking d in the cache of size C at time t.

In a random sampling-based 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:

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

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 $Q_{C,K}(x == x_{\rho(r)})$.

We propose KRR based on an approximation of 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, $\rho_{t,i}(r(S_t(i))) = i$.

Based on the 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 1, 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:

$$maxPriority(y_t(i-1), S_{t-1}(i)) = \begin{cases} S_{t-1}(i) & random(0,1) < \left(\frac{i-1}{i}\right)^K \\ y_t(i-1) & otherwise \end{cases} \tag{2}$$

With $maxPriority$ function defined, one can trivially simulate the KRR replacement scheme using Mattson's linear stack update procedure described in Section 2.1.

Figure 2 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 the less recent one ranks lower; On the other hand, the KRR replacement policy maintains object's ranking explicitly through the 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.

According to the KRR algorithm, an object on stack position i, denoted as $S_t(i)$, will be evicted from the 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 the 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 the same approach we show that:

$$\begin{aligned} &\Phi_{C,K}(S_t(i)) = \\ &\left(\frac{i^K - (i-1)^K}{i^K}\right) * \left(\frac{i}{i+1}\right)^K * \left(\frac{i+1}{i+2}\right)^k * \ldots \left(\frac{C-1}{C}\right)^K \\ &= \frac{i^K - (i-1)^K}{C^K} \end{aligned} \tag{3}$$

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 the 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 downshift their ranking by one, and their relative ranking to one another remains the same. Unlike K-LRU, KRR performs one-way shifts of object's rank only on a subset of objects as illustrated by Figure 2. Although less recent objects are still likely to rank lower in KRR cache, due to these probabilistic shifts, objects' ranking in KRR cache does not fully resemble 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 accessing objects with the same recency order, i.e. loop

pattern. To further reduce the error, we make a simple modification to 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, the 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 objects 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.

3.2 Backward Stack Update

As described in Section 2.1, the naive stack algorithm requires $O(M)$ update time for every access. A downshift object would need to be compared to and swapped with the objects from the stack top to the recent hit location based on the *maxPriority* function in Equation 2. Clearly, $O(M)$ per update is prohibitive for online processing. To overcome the expensive update overhead, we propose an efficient backward stack update mechanism, which reduces the overhead from $O(M)$ to $O(logM)$.

First, we notice that the probability that $maxPriority$ function returns $S_{t-1}(i)$ increases as we scan down the stack. 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*.

Let β_{swap} denote the total number of swap positions per stack update, then the expectation is: $E(\beta_{swap}) = O(KlogM)$, 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 the 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 the entire stack.

Mattson et al.'s linear stack update determines swap positions by performing random draws from stack top till $S_{t-1}(\phi)$. We find that a more efficient way can be done by generating swap positions backward, 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 3, we know the eviction probability of an object in the KRR cache is directly associated with its stack position. Furthermore, the cumulative distribution function (CDF) of Equation 3 is $P(X \leq x_i) = \left(\frac{i}{C}\right)^K$. Now, we can obtain v_β by simply taking the inverse of the CDF with $C = \phi - 1$. For $v_{\beta-1}$, since v_β is already identified, we can compute it using a similar idea with $C = v_\beta - 1$. Algorithm 1 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 [42].

The expected running time for Algorithm 1 is $O(logM)$, because the expected number of swap positions is bound by $O(logM)$, and each iteration of Algorithm 1's while loop computes exactly one swap position. By using Algorithm 1, our KRR model can approximate the K-LRU cache in just $O(NlogM)$ time.

Algorithm 1 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
```

3.3 Variable Object Size-Aware KRR

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 as discussed in Section 2.5.

The basic array implementation of the KRR stack in Algorithm 1 implicitly assumes that all objects on the stack have identical sizes. The stack distance can be directly related to the object's array index. However, for workloads with diverse object size distribution [32], computing the object's stack distance based on its logical location on the stack could be problematic. For example, in Figure 3, we see that under uniform object sizes assumption, the estimated byte-level stack distance of object D is 16 which significantly differs from the actual byte-level stack distance 11. In other words, the predicted cache size to achieve a cache hit when referencing D is 16, but in reality, a cache size of 11 is sufficient. This highlights the importance of considering object size distribution when modeling cache performance. However, calculating the precise byte-level stack distance is costly, as it requires summing up the sizes of all objects from the stack top to the referenced object.

To collect byte-level stack distance efficiently, our solution is to add an additional array structure, *sizeArray*. The length of *sizeArray* is $\log_b M$, where M is the stack size. Each entry of the *sizeArray* maintains a partial accumulation of stack size, specifically, entry i of the *sizeArray* (Σ_i) stores the total size of objects from stack top to stack position b^i, where b is the base parameter. Figure 4 illustrates the stack update process for a KRR stack with a base-2 *sizeArray*. As depicted in Figure 4, to compute byte-level stack distance of the object in the stack position Φ, we use the actual cumulative

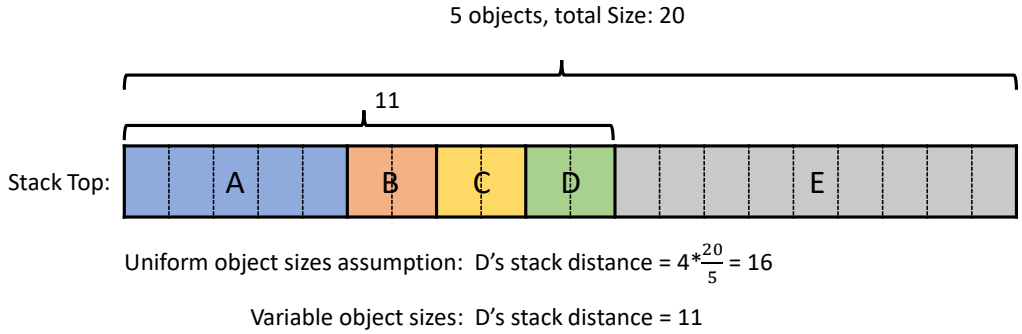


Fig. 3. Byte-level stack distance example

size to position 64 (Σ_{64}) and add it with the approximated size from position 64 to Φ ($(\Sigma_{128} - \Sigma 64) * \frac{\Phi-64}{128-64}$) through interpolation. Once we obtain the byte-level stack distance, we update the stack similar to the Algorithm 1, and we also update all Σ_i where $i < \Phi$ as illustrated. Since the length of *sizeArray* is logarithmically bounded with respect to the KRR stack length, the cost of maintaining 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 2. Now we are capable of constructing MRC of variable object-size K-LRU cache.

Algorithm 2 Byte-level KRR Stack Distance

procedure StackDistance(st, $sizeArry$, b, ϕ)
  ▷ st: Ordered Stack, implemented as an arrayList
  ▷ sizeArry: array of partial stack sizes
  ▷ b: sizeArray's base
  ▷ ϕ: stack position of referenced object

  $index \leftarrow \log_b(\phi)$
  $sdLow \leftarrow b^{index}$
  if $sdLow < \phi$ **then**
    $sdHigh \leftarrow b^{index+1}$
    $res \leftarrow (sizeArry[index+1] - sizeArry[index]) * \frac{\phi - sdLow}{sdHigh - sdLow}$
  else
    $res \leftarrow 0$
  end if
  return $sizeArry[index] + res$
end procedure

4 kRedis: K-LRU Merged MRC Guide and Latency-Aware Partitioning

In this section, we focus on the multi-tenant cache use case and propose a novel multi-tenant memory allocation scheme that is guided by the merged K-LRU MRCs and dynamically configure the sampling size K for each individual tenant to better explore workload locality.

When multiple applications/tenants share a single Redis cache instance, the available memory is partitioned for each tenant to meet their caching requirements. Since Redis memory space is limited, maximizing the utilization of the shared memory pool is critical for system performance.

In existing MRC-guided partitioning designs, such as LAMA [5], mPart [22] and pRedis [24], reference keys are randomly sampled to construct reuse time histogram, from which an MRC for each application can be calculated using the footprint model or the AET model [6], [21]. At a specified interval measured as the number of requests, a dynamic programming algorithm is invoked to minimize the number of expected misses based on the constructed MRC. This partitioning scheme is based on the fact that the MRC of each tenant is fixed, which is true for Memcached when applying the exact LRU eviction policy. However, for Redis and other caches that employ K-LRU, the sampling size K's impact on miss ratio can be significant. This means there are multiple MRCs available corresponding to different K for each application. Therefore, the search space for finding the optimal partition solution significantly increases.

We introduce a memory partitioning scheme for K-LRU Redis cache named $kRedis$ assuming that K is configurable on the fly. To reduce the search space, kRedis dynamically allocates memory based on a merged MRC from a small set of MRCs of different Ks. The MRCs of each K-LRU cache for each tenant application are constructed online using the KRR model introduced in Section 3.3 and spatial sampling described in Section 2.7.

4.1 Merged MRC

Intuitively we need the MRCs with different Ks for each application, where we can choose the MRC_K that yields the minimum miss ratio at cache size C and record the corresponding sampling size K. In order to avoid expanding search space, we simply merge several MRC_Ks of each tenant into one MRC where only the lowest miss ratio and respective K at every cache size are recorded. For each tenant, miss ratio curves MRC_K with different K are merged into a single miss ratio curve $mMRC$ as following:

$$mMRC(C) = min\ MRC_K(C) \tag{4}$$

Now the optimization problem can be reduced to a partitioning problem with each tenant having one fixed MRC, $mMRC(C)$. Figure 5 shows three K-LRU MRCs of MSR src1 with K = 1, 5, and 16 merged to one $mMRC$. The sampling size K is not shown on $mMRC$, it is encoded in

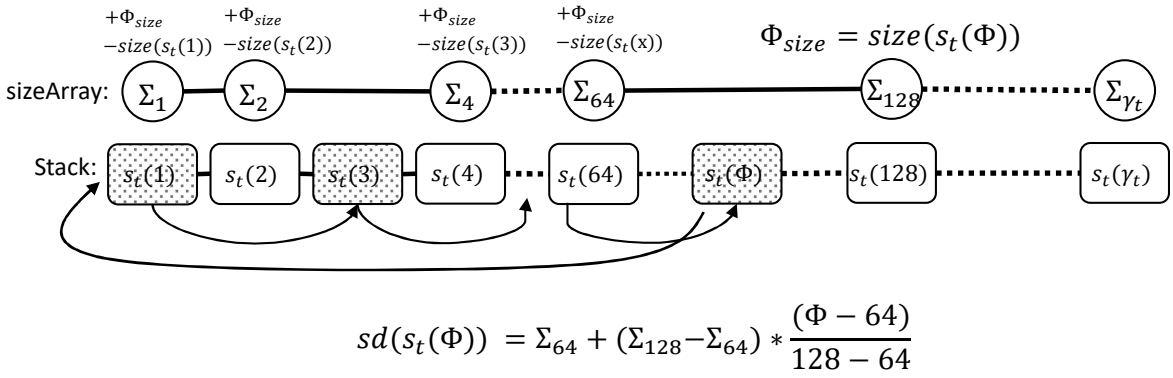


Fig. 4. Variable object size stack update

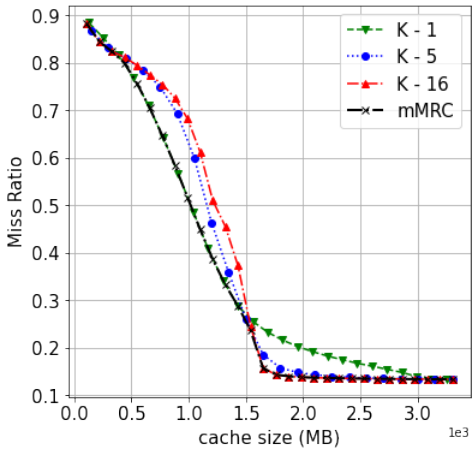


Fig. 5. Merge K-LRU MRCs to $mMRC$.

the data structure of $mMRC$ so that later it can be used to configure the eviction policy of the corresponding tenant.

4.2 Memory Partitioning

Similar to mPart and pRedis [22], [24], a memory partitioning scheme is computed at the end of each periodic interval window, which is preset as the number of requests (default is 1 million). At each evaluation interval, for each tenant, we measure its average miss latency and access rate, and construct the K-LRU MRCs by spatial-sampling its requests. We then determine the memory partitioning scheme using a dynamic programming algorithm. The partitioning is then enforced at the next interval. Specifically, there are four steps in each evaluation interval.

Step 1: Latency & Access Rate Measurement

We record the cumulative miss latency and the number of cache misses to calculate the average miss latency p for every application sharing a memory pool. The tenant access rate a is estimated as the rate of interval time and tenant access count in the interval window.

Step 2: Merged K-LRU MRC Construction

For each tenant, we construct MRC_K for several small Ks on the fly based on spatial sampling and the variable object size-aware KRR model, then derive the merged $mMRC$ for each application as discussed in Section 4.1.

Step 3: Memory Partitioning Scheme and K Selection

To take the impact of miss latency into account, we estimate interval miss latency P_i for application i as follows.

$$P_i = mMRC_i(C_i) * a_i * p_i \tag{5}$$

Our goal is to minimize the overall miss latency for a set of N tenant applications in a Redis cache instance with total memory M.

$$min \sum_{i=1}^{N} P_i = \sum_{i=1}^{N} mMRC_i(C_i) * a_i * p_i \tag{6}$$

$$subject\ to \sum_{i=1}^{N} C_i = M$$

Inspired by DCAPS [8], kRedis memory partitioning could achieve various optimization targets by adjusting Equation 6. For example, the following metric can be adopted to optimize hit throughput which is defined as the number of GET hits per access time.

$$max \sum_{i=1}^{N} TP_i = \sum_{i=1}^{N} (1 - mMRC_i(C_i)) * a_i \tag{7}$$

The optimization problem of Equation 6 can be solved using dynamic programming similar to mPart [22] and pRedis [24].

At the end of each evaluation interval, we run the memory allocation algorithm presented in Algorithm 3. We calculate the minimum overall miss latency and record the memory allocation for each application. The i loop (line 8) and j loop (line 9) combined find the best latency when the first i tenants are allocated j amount of memory. The innermost C loop (line 10) enumerates all possible allocations of tenant i subject to the upper bound of memory size j (line 10). The second loop nest from line 21 to 24 backtracks optimal memory partition recorded in $\{A\}$.

The time complexity of such dynamic programming is $O(VM^2)$, where V is the number of applications and M is the size of the memory pool. In real applications, the memory bound M could be a large value in bytes, but we use configurable larger granularity G in memory allocation, for instance, 1 MB or 10 MB, according to the application

profiles. Then the time complexity becomes $O(V(M/G)^2)$ which is affordable for online usage.

Algorithm 3 Memory Allocation

Require: M ▷ Total cache memory
Require: {V} ▷ Set of tenants
Require: {mMRC} ▷ Set of merged MRC for each tenant
Require: {a} ▷ Set of access rate for each tenant
Require: {p} ▷ Set of average miss latency for each tenant

procedure ARBITRATE
  for $i \in V$ **do**
    for $j \leftarrow 0$ to M **do**
      $f[i][j] \leftarrow \infty$
    end for
  end for
  $f[0][0] \leftarrow 0$
  for $i \in V$ **do**
    for $j \leftarrow 0$ to M **do**
      for $C \leftarrow 0$ to j **do**
        $miss_i \leftarrow mMRC_i(C) * a_i$
        $latency \leftarrow f[i-1][j-C] + miss_i * p_i$
        if $latency < f[i][j]$ **then**
          $f[i][j] \leftarrow latency$
          $Target[i][j] \leftarrow C$
        end if
      end for
    end for
  end for
  $T \leftarrow M$
  for $i \leftarrow N; i \rightarrow 1$ **do**
    $A_i \leftarrow Target[i][T]$
    $T \leftarrow T - Target[i][T]$
  end for
  return $\{A\}$
end procedure

Step 4: Dynamic Memory Allocation and K Adjustment

Once the memory partitioning scheme is determined, Redis memory should be allocated for each application accordingly. Inspired by the work of pRedis [24], we maintain two arrays to book-keep the amount of memory used in each tenant and the suggested memory allocation by our model. To ensure tenant references are processed under its appropriate K-LRU eviction policy, we configure the sampling size K_i for application i according to $mMRC_i$.

We adjust the tenant memory usage by modifying the Redis eviction process. Initially, Redis memory size is maintained by the eviction procedure named freeMemoryIfNeeded. Each time Redis receives a request, this procedure checks the used memory against the max-memory setting and free items if needed. In kRedis, we pick the application in which the used memory is greater than the suggested memory for eviction. Thus the tenant space is adjusted on the object level and the pace of adjustment is dependent on the tenant's request pattern.

4.3 Efficient Random Sampling Eviction Design

The challenge in step 4 of Section 4.2 is that from the whole Redis key space, how can we effectively sample K keys that belong to a specific tenant. pRedis [24] adopts a bloom filter to determine the tenant belonging of each sampled key in the process of eviction. However, the bloom filter time overhead can be notable according to our evaluation. Each time a key is stored in the cache, the bloom filter needs to check if such key is a new key, then map the key to its tenant in the bloom filter structure. On evictions, every randomly sampled key must be checked to decide if it belongs to the memory-overusing tenant. According to our evaluation, when there are 4 tenants, the key space size of each tenant is about 12 million, and the cache max-memory is set to 50% of the working set size, the total time used in the eviction process is about 50 seconds, in which the bloom filter judgment time takes 15 seconds or 30%. On average, the bloom filter judgment takes 2 μs to identify a key for eviction. If the number of tenants and key space increase, the bloom filter time overhead in the K-LRU eviction process will only be larger.

Our solution is to separate key dictionaries for the tenants in Redis. The original Redis design uses a single dictionary, where the keys of all the tenants reside, so there is no ready-made support for multi-tenant memory partitioning in a single Redis instance. By using multiple dictionaries, the K-LRU eviction is to simply sample K keys randomly in the desired tenant's dictionary and the bloom filer is no longer needed in both setting and eviction processes. In practice, it is trivial to distinguish tenant keys using the unique client ID embedded in the Redis data structure or other available parameters such as the source socket id of a request.

The data objects of Redis are stored as dictionary entries in the hash table and connected by pointers. Our multi-dictionary design only sets up multiple tenant-wise hash tables pointed by the dictionary header, which has a negligible effect on the Redis command processing, and the overhead is the metadata of the dictionary header, which is in a total of 176 bytes per tenant.

5 IMPLEMENTATION

Unlike the LRU stack, the KRR stack only shifts a small subset of objects on the stack per stack update. To take advantage of that, we implement 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 the algorithms described in Section 3.2. Then we perform cyclic swapping on all marked positions. In our implementation, we adopt the spatial sampling technique described in Section 2.7. By default, we use a sampling rate of $R = 0.001$, but to ensure the accuracy of spatial sampling using SHARDS [3], a higher sampling rate of $R = 0.01$ is applied to workloads with relatively small working set sizes (less than 8M distinct objects).

For MRC accuracy evaluation and comparison, we have implemented Mattson's LRU stack algorithm using a balanced search tree [34]. 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 exact LRU policy. We also implement SHARDS [3], which can output an approximated MRC for the exact LRU policy. To reveal the ground truth of the miss ratio of a K-LRU cache, we have 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 run the simulator multiple times for different cache sizes and use interpolation for miss ratio prediction.

For performance evaluation, we implement kRedis on top of Redis-4.0 with the default Jemalloc allocator. It uses KRR to model K-LRU policy with random sampling size K of 1, 5, 8, and 16, and chooses the best K on the fly for each tenant. It adopts the multi-dictionary scheme to accelerate tenant-level K-LRU random sampling and eviction process. Besides the original Redis as a primary baseline, we use pRedis [40] as a secondary baseline to evaluate the performance of kRedis in multi-tenant key-value cache. pRedis is based on the original Redis single-dictionary design and uses EAET to model exact LRU plus a bloom filter to discriminate key-value's tenant belonging. Both Redis and pRedis set K to 5 as default.

6 Experimental Evaluation

In order to evaluate the effectiveness of kRedis, we first give a brief description of the experimental setup and evaluation workloads. Second, we evaluate the accuracy of the predicted miss ratio of KRR. Third, we compare the performance of Redis, pRedis, and kRedis. Next, we discuss the memory size impact, throughput, tail latency, and both time and space overhead of our design. Finally, we compare kRedis with our previous work DLRU [27] and discuss the applicable fields of those two schemes.

6.1 Experiment Setup

We use two separate machines for our evaluation. Machine A 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. Machine B is configured with Intel(R) XEON(R) E5-2620 v4 2.10GHz processor with 20 MB shared LLC and 128 GB of memory, and the operating system is Ubuntu 18.04.6 LTS. All major evaluations are done on machine A, machine B is only used in Section 6.4.3.

A Redis cache server and multiple tenant front-ends are deployed on the local host. We implement Redis tenant front-end based on Hiredis library [45], which reads references from an evaluation trace and sends access requests to the Redis server on the fly. When the Redis server returns a miss, the tenant front-end will immediately follow a SET command to store the key-value pair into the server. With such a setup, the miss latency is simply the round-trip setback time between the tenant front-end and the Redis server. In most of our evaluations, we use various time delays to simulate fetching items from the database, providing flexibility and variety in the miss latency setup to our evaluation. We also set up an evaluation environment with a real remote and local database to cross-validate the test case with the simulated environment. When evaluating multi-tenant scenarios, we set up multiple tenant front ends, with each tenant front end repeatedly sending requests from a workload until the server terminates.

6.2 Workloads

We use two different workloads for our evaluation:

- **MSR** MSR Cambridge suite [29] is a collection of block-level I/O traces from 36 volumes across 179 disks on 13 different enterprise data center servers in a Microsoft data center. We evaluate our model on all 13 traces, as well as the merged "master" MSR workload which is also used in Waldspurger *et al* [3]. The workloads encompass various applications such as home directories, project directories, hardware monitoring, firewall/web proxy, source control, web staging, media services, and more. The 13 workloads' reference counts range from 1 to 181 million, working set sizes range from 1 to over 1000 GB. More detailed trace information can be found in [30].
- **Twitter** Twitter cache traces [32] is a collection of one-week-long cache request traces from 54 Twitter's in-memory caching clusters. We use sub-traces from Twitter clusters to evaluate our K-LRU model and kRedis performance. Each sub-trace consists of 100 million requests, detailed information of each workload including working set size, object size distribution, compulsory miss ratio, etc. can be found in [32], [33].

6.3 MRC Accuracy

To measure the accuracy of KRR model, we report the mean absolute error (MAE) as used in [3]. 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 of $R = 0.01$ to those workloads with a small working set size (less than 8M objects in our experiments). (3) Modeling error. Since KRR only orders objects according to their recency at a coarse granularity, KRR and K-LRU are not statistically identical except when $K = 1$.

We evaluate KRR with a uniform object size version and the variable object size-aware version using MSR and Twitter traces, where object sizes vary. We compare the modeled MRCs with the actual MRCs generated from directly simulating the K-LRU cache under 40 different cache sizes that are evenly distributed over the workload's working set size. For convenience, we use *uniKRR* and *varKRR* to denote the uniform object size KRR and the variable object size-aware KRR, respectively.

The *uniKRR* does not take an object's size into consideration, which leads to unreliable MRC results when workloads

TABLE 1
MAE under different sampling size for variable size MSR and Twitter workloads

	varKRR		varKRR+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

contain diverse object sizes. In Figure 6, we show MRCs from 8 different representative traces (4 MSR and 4 Twitter). Each graph compares uniKRR and varKRR with the true MRC generated by the K-LRU simulator. We observe that the MRCs generated based on uniform size assumption (uniKRR) do not always approximate the real MRCs well (shown in Figure 6(A)). In contrast, varKRR approximates the real MRCs with negligible errors. And varKRR's time cost stays at the same level as uniKRR. Table 1 summarizes the MAE of MSR and Twitter traces under different K values from 1 to 32 for varKRR. Overall, varKRR 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.

6.4 Access Latency

The memory allocation objective in Equation 6 is to minimize the overall miss latency or response time, so we use the average of tenants' mean access latency as the evaluation metric. The access latency is the wall clock time used by each access. We use the MSR and Twitter workloads in this evaluation. Redis cache maximum memory is set to 50% of the total working set size.

The workload sensitivity to the changes in sampling size K is the key to the exploration of the K-LRU miss ratio gap, and it is also the source of potential performance improvement from pRedis to kRedis. In order to compare the performance between the two, we first choose the workloads that are sensitive to the change of K and conduct case studies on a 4-tenant system. The MRCs of some example workloads are shown in Figure 7. We then stress the system by increasing the number of tenants to 15 using randomly selected traces from the Twitter suite.

6.4.1 4-Tenant Case Study

In this case study, we set up four tenants with MSR web and src2 workloads, representing fetching objects from web local-DB, web remote-DB, src2 local-DB, and src2 remote-DB, respectively. According to the access latency analysis of real Redis traces [40], the remote DB miss latency is typically distributed around 2000 μs. Therefore the miss latency is configured to 200 μs (local-DB) and 2000 μs (remote-DB), respectively. As shown in Figure 8, when compared to Redis, pRedis reduces the mean access latency of both web remote and src2 remote at the cost of a slight increase of the latency with the local-DB ones. And compared to pRedis, kRedis has further reduced the latency of web remote. Overall, the average access latency improvement of kRedis is 17.3% and 14.3% compared to Redis and pRedis, respectively.

Furthermore, in order to analyze the contribution of each optimization within kRedis, including the multi-dictionary design, the KRR model, and dynamic K configuration based on merged MRC, we use a series of variants to decompose their impact on performance, results are shown in Figure 9. First, pRedis, which considers miss latency, shows a 3.6% improvement over Redis, contributed by locality- and latency-aware memory partitioning. Second, we transform pRedis to the multi-dictionary design, which gains an additional 1.5%. Third, we use varKRR instead of EAET to model MRC under a fixed K of 5, which is the default setting in Redis and pRedis. Note that pRedis uses EAET to model exact LRU rather than K-LRU with $K = 5$, which could reduce MRC accuracy. With the more accurate KRR model, this variant of pRedis shows a 10.8% of improvement over Redis. Lastly, kRedis, combining optimization of KRR, dynamic K configuration, and multi-dictionary design, yields a 17.3% improvement over Redis. When comparing kRedis to the pRedis variant with varKRR and fixed K of 5, the dynamic K configuration contributes a 6.5% improvement.

In Figure 8, we observe that compared to pRedis, kRedis has decreased the access latency of web remote. To dig deeper into the process of dynamic K selection based on merged MRC, we provide snapshots of web remote K-LRU MRCs constructed by KRR in two individual intervals. Figure 10 (A) shows that in the 11th interval, the K-LRU MRC with $K = 1$ yields a lower miss ratio than other Ks at the partitioned memory of 800 MB, and kRedis sets K to 1 for the tenant. Figure 10(B) shows that in the 15th interval, the K-LRU MRC with $K = 16$ shows a lower miss ratio than other Ks at the partitioned memory of 800 MB, and kRedis sets K to 16 for the tenant.

In another test case, we use Twitter sub-traces of cluster4.0, cluster29.0, cluster34.0, and cluster54.0 to generate references of 4 tenants, with their miss latency configured to 200 μs, 400 μs, 1000 μs, and 2000 μs, respectively, bringing more variety to the simulated miss latency. The average access latency improvement of kRedis is 27.0% and 16.7% compared to Redis and pRedis, respectively. Figure 11 summarises the latency results of pRedis, pRedis variants, and kRedis, compared to Redis baseline. First, we observe a 12.4% improvement related to locality- and latency-aware memory allocation. Second, with the multi-dictionary design, the improvement increases to 14.5%, Then equipped with varKRR for K-LRU MRC construction of K fixed to 5, it shows an 18.2% decrease in latency. Finally, kRedis which employs all optimizations shows a 27.0% improvement against Redis.

6.4.2 15-Tenant Case Study

To evaluate the performance of kRedis for a large number of tenants, we increase the number to 15, adding pressure to MRC construction, the cache partitioning algorithm, and the sampling-based eviction process. Now we assume each Twitter cluster trace comes from multiple tenants. Each reference of Twitter clusters includes a parameter named Client-ID which is the anonymous front-end service client who sends the request. We use this ID modulo 15 to generate a tenant ID. We randomly select 6 traces from Twitter:

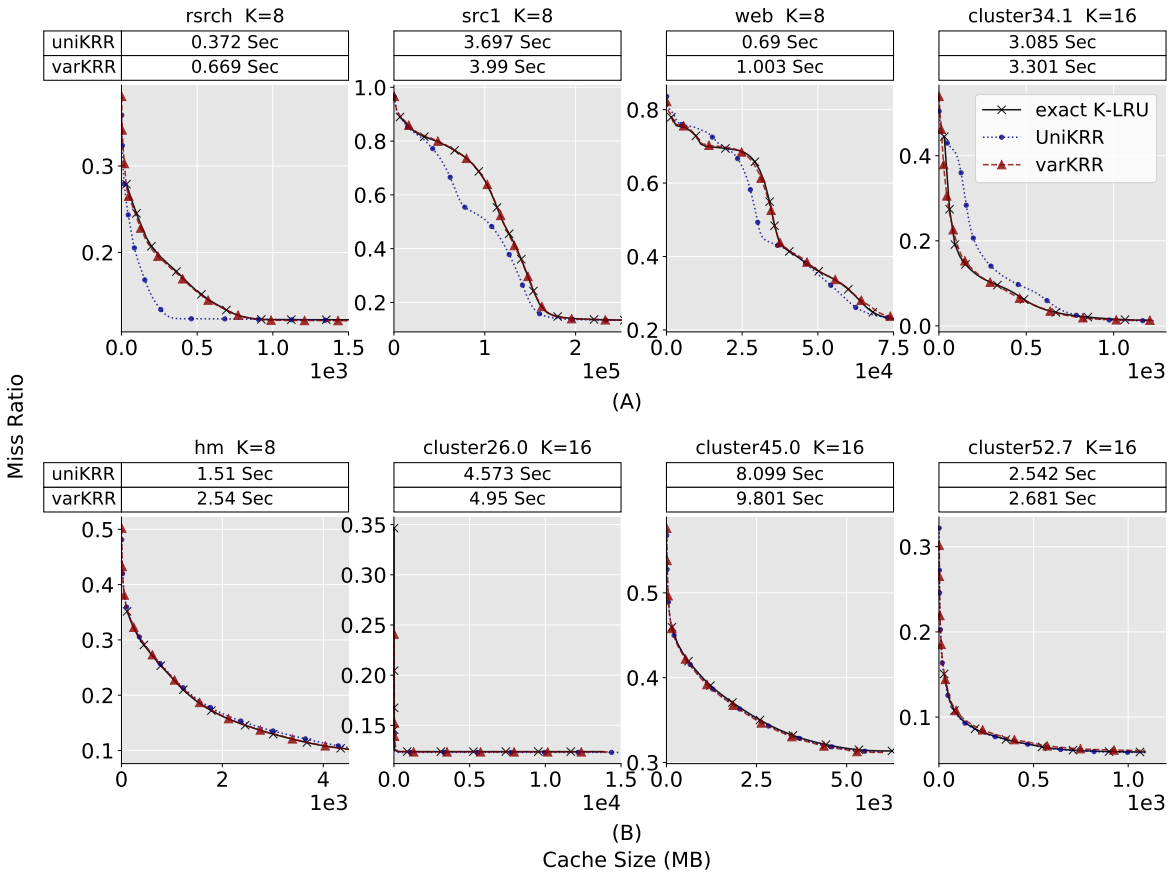


Fig. 6. Accuracy and time cost for uniKRR and varKRR

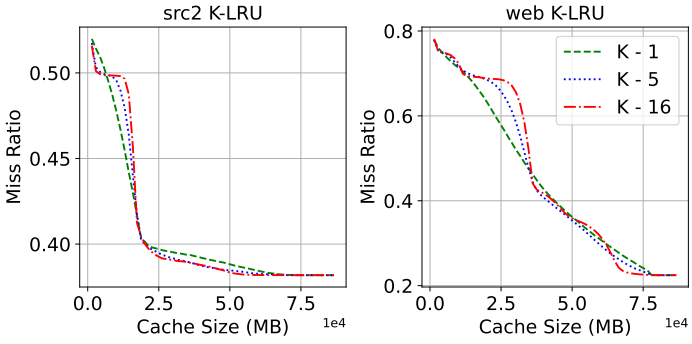


Fig. 7. Workloads that are sensitive to the change of K. Miss ratio show gaps between different Ks at the same cache sizes.

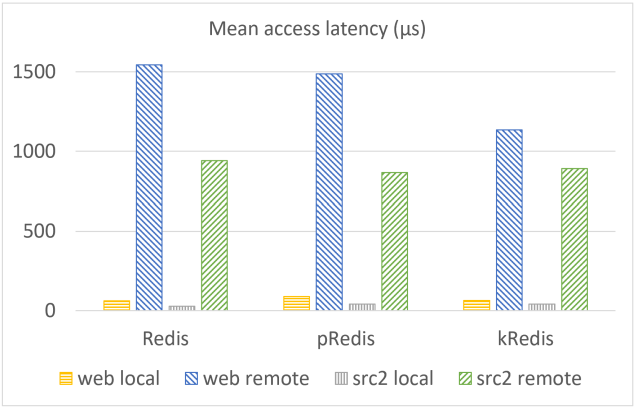


Fig. 8. Average access latency reduction with 4 tenants loading MSR workloads.

cluster4.0, cluster17.0, cluster18.0, cluster29.0, cluster44.0, and cluster52.0. For each workload, we set up the miss latency of each tenant based on the observed exponential distribution of miss latency of real-world Redis traces [40]: the miss latency of each tenant is set as 1 μs, 2 μs, 4 μs, 8 μs, 16 μs, ..., 4,096 μs, 8,192 μs and 16,384 μs, respectively. Figure 12 shows the average latency reduction of pRedis and kRedis, compared with the Redis baseline. kRedis reduces access latency up to 50.2% against Redis, and improves up to 24.8% when compared to pRedis.

6.4.3 *Real Back-End Database Case Study*

In this section, we use two separate machines described in Section 6.1 running real back-end MySQL databases to cross-validate the test case shown in Section 6.4.1 with MSR workloads. Both Redis cache server and tenant front-end are running on Machine A. The web local-DB and src2 local-DB are also running on Machine A, and the web remote-DB, src2 remote-DB are running on Machine B. Both local and remote DB are deployed on MySQL Community

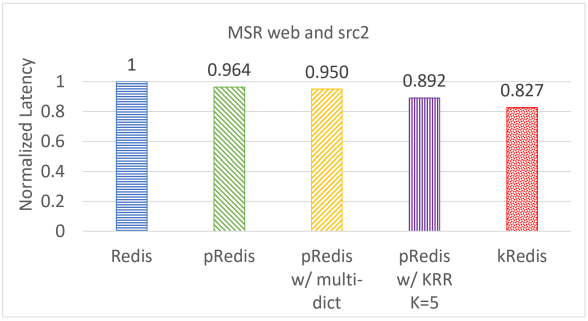


Fig. 9. Impacts of kRedis optimizations on latency for MSR workloads, compared to Redis baseline.

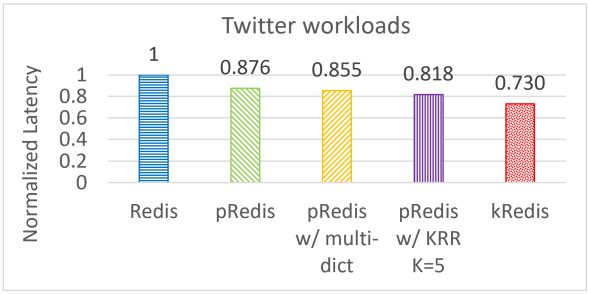


Fig. 11. Impacts of kRedis optimizations on latency for Twitter workloads, compared to Redis baseline.

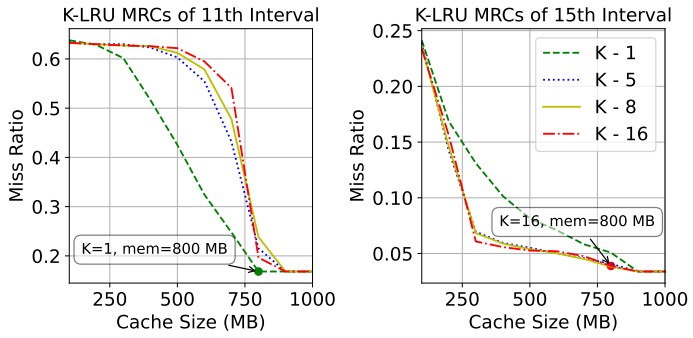


Fig. 10. K-LRU MRCs of web remote in two evaluation intervals and K configurations. The periodical evaluation interval size is 1 million requests.

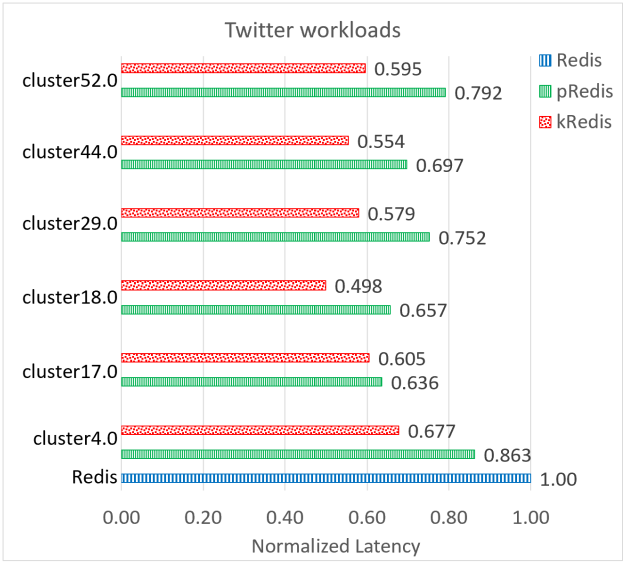


Fig. 12. Average access latency reduction in Twitter workloads compared to Redis baseline. Tenant accesses are generated by partial references from a workload distinguished by Client-ID.

Server 8.0.33. In this real MySQL back-end DB setup, the miss latency of the local DB is distributed around 200 μs, and that of the remote DB is distributed around 2000 μs. Compared to Redis and pRedis, the average access latency improvement of kRedis is 17.2% and 13.3%, respectively. The results show no notable difference between the real back-end database and the simulated back-end database.

6.5 Impact of Memory Size

The max-memory setting of Redis impacts cache performance. Over-provisioned memory to a cache can lower the miss ratio of cache but at a higher cost on DRAM. In contrast, over-tight memory provision brings harm to cache performance. Intuitively, a tighter memory size introduces a higher miss ratio for all tenants, where kRedis has the potential to dynamically partition memory to meet the space requirement of tenants that are performance-critical for the optimization target. We evaluate kRedis performance on different max-memory settings to observe this trend based on the 4-Tenant test case of Twitter workloads in Section 6.4.1. The result shows that, compared to Redis, kRedis improves average access latency by 21.2%, 27.0%, and 49.5% with max-memory as 75%, 50%, and 25% of the working set size, respectively. The tighter limit on Redis memory brings higher performance improvement of kRedis against Redis.

6.6 Throughput

As described in Section 4.2, kRedis could achieve different optimization targets. We adopt Equation 7 and use the rate of reference hits against system time as the metric to evaluate throughput. We demonstrate the effect of kRedis with a 4-tenant case study. The four tenants load MSR workload mds, src2, stg, and web respectively. Cache max-memory is set to 50% of the working set size. All tenants' miss latency is set to 2000 microseconds simulating fetching objects from remote DBs. Table 2 shows kRedis improves the average throughput by 262.8% and 61.8% compared to Redis and pRedis, respectively. The similar setup for the 4-tenant case using 2 web and 2 src2 workloads shows similar results.

TABLE 2
Throughput (hits/sec) in MSR workloads

	mds	src2	stg	web	avg
Redis	1113	1455	2499	1701	1692
pRedis	1474	9839	1949	1920	3795
kRedis	485	22071	1485	515	6139

6.7 Tail Latency

kRedis has been proven to improve tenants' average latency as well as hit throughput, but we still need to figure out if the statistics tracking and memory allocation of kRedis affects references' latency disproportionally. We evaluate the tail latency of kRedis in the first 4-tenant case study with MSR workloads discussed in Section 6.4.1. Table 3 shows the results. When comparing kRedis with Redis, there are no significant differences between the two for tenants with remote DB. However, since kRedis allocates more memory to the remote ones at the cost of increasing the miss rate of local ones, kRedis shows higher latency than Redis for the response times of src2-local.

TABLE 3
Request tail latency(μs)

		90th	95th	99th	99.9th
web-remote	Redis	2044	2049	2066	2457
	kRedis	2046	2056	2100	2270
src2-remote	Redis	2040	2047	2062	2129
	kRedis	2044	2054	2091	2219
web-local	Redis	223	228	241	267
	kRedis	224	231	253	307
src2-local	Redis	32	38	51	255
	kRedis	44	203	247	284

6.8 Time and Space Cost

In this section, we evaluate the time and space overhead of our approach. There are two sources of time cost: K-LRU MRC modeling with spatial sampling, and hash table resizing with multi-dictionary design in kRedis. The space cost also comes from two aspects: the implementation of the KRR stack and the multiple dictionaries. First, we discuss time and space costs related to the KRR model. Then we analyze the overhead of multi-dictionary design.

To measure the efficiency of our KRR model and stack update mechanisms, we compare backward stack update methods (with/without spatial sampling) with the naive linear stack update method and the simulation/interpolation approach. We simulate K-LRU under 25 different cache sizes evenly distributed across its working set size. For demonstration purpose, we use the first one million references from MSR src1 trace, and set $K = 5$. Table 4 is a summary of the results. We see that the backward stack update method shows an 8247 times improvement over the linear stack update approach. When spatial sampling with $R = 0.01$ is applied, the running time is further improved by two more magnitudes. We also observe similar time cost improvement on other workloads.

Next, we use the merged "master" MSR trace to compare the running time of KRR+Spatial sampling with the existing LRU MRC approximation technique, SHARDS. Table 5 contains the running time for backward stack update KRR and SHARDS. The running time of KRR shown in Table 5 is the average across different Ks $(1, 2, 4, 8, 16, 32)$. The average running time for KRR with backward stack update and SHARDS is very close to the master trace in our test.

With the previous 15-tenant case study in Section 6.4.2, for all the 6 evaluated workloads, the total KRR time overhead including reference tracking and K-LRU MRC modeling is in the range from 0.57% to 0.66% of total execution time.

TABLE 4
Running Time Comparison for Processing One Million MSR src1 Requests

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

TABLE 5
Master Trace Comparison

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

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 the hash table and other auxiliary entries. For variable object size-aware 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 is just 0.036% of the working set size.

In the following, we discuss the time and space overhead of multi-dictionary design in kRedis. The Redis hash table is capable of resizing itself according to the load factor. In the process of expansion or contraction of a hash table, Redis performs rehash operations which bring extra time overhead. In the original Redis, once the maximum memory is reached, the size of the single hash table is generally stable. But our multi-dictionary design may bring extra overhead while the size of each tenant's key space is changing according to the dynamic memory allocations. To measure the efficiency of our multi-dictionary design, we profile the time overhead of hash table resizing and compare it with Redis. We use the Twitter cluster 54.0 workload and emulate a 16-tenant system as described in Section 6.4.2. Redis' total rehashing takes 1 second out of 81199 seconds of running time, while kRedis' total rehashing takes 9 seconds out of 44474 seconds of running time. The rehashing time cost of multi-dictionary design is almost negligible as it accounts for only 0.02% of total running time.

2. Many real in-memory cache workloads have much higher average key-value size [32]

In Redis, all key-values are stored as dict-Entry in the hash table, which is organized in a dictionary header structure containing type, pointer to hash table, rehash index, etc. In kRedis' multi-dictionary design, all space costs of actual key-value pairs are the same as Redis, the only overhead is the metadata of the dictionary header, which is in a total of 176 bytes per tenant. Using the same instance in evaluating the space overhead of the KRR stack, assuming there are 15 tenants, then the space overhead for processing workload with 100 million distinct objects is just 1.32e-5% of the working set size.

6.9 kRedis vs DLRU

As described in Section 2.3, DLRU [27] is also capable of reducing the overall access latency of a single-tenant, fixed memory size key-value cache by using miniature cache [28] to simulate the behavior of various K-LRU caches and explore the potential miss ratio gap of various sampling size Ks. kRedis can be downgraded to handle a single tenant, where KRR is applied to identify an optimal K. In this section, we first use a single-tenant environment to compare kRedis and DLRU. Then we extend DLRU to construct tenant MRCs in a multi-tenant key-value cache and use those to guide tenant memory partitioning. We use a multi-tenant test case to discuss the limitations of DLRU versus kRedis.

6.9.1 Single-Tenant Case Study

In this case study, we set up a fix-memory size Redis instance running a single tenant loading MSR workload. We use KRR and DLRU to guide the selection of K, respectively. We conduct two tests, using MSR workload src1 and web, respectively. We choose the Redis memory size as 30% of the workload's working set size where the miss ratios of various random sampling Ks show a large gap. The tenant's miss latency is configured to 2000 μs to represent fetching objects from the remote database. As shown in Figure 13, there is no notable difference in performance between DLRU and kRedis in the single-tenant use case. This indeed verifies the accuracy of the KRR model against simulation.

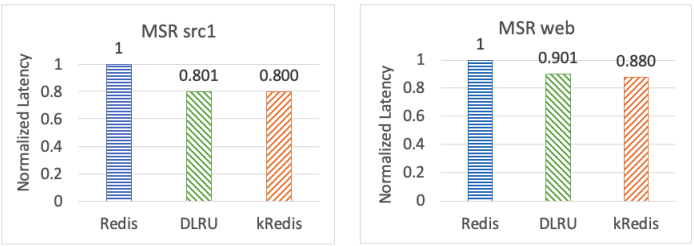


Fig. 13. Average access latency reduction of DLRU and kRedis with single tenant loading MSR workload. There is no notable difference between the two schemes.

6.9.2 Multi-Tenant Case Study

To adopt DLRU for multi-tenant partitioning, we extend DLRU by generating MRCs through miniature cache simulation and interpolation. For each tenant, a selection of Ks, and a selection of cache sizes, we use D-LRU miniature cache to find the miss ratios. We then construct an MRC through interpolation for each K of each tenant, and use these MRCs to guide partitioning instead of the KRR MRCs. It is worth noting that interpolation may not be able to capture every inflection point on the MRC, thus losing accuracy in the constructed MRC.

To compare the extended DLRU against kRedis, we employ the same 15 tenants set up as the one used in Section 6.4.2 with Twitter cluster18.0 workload. In this test case, for each of the 15 tenants, we set up 20 cache sizes uniformly distributed across the range of Redis max-memory and provide 4 K options. Therefore, there are a total of $15 * 20 * 4 = 1200$ independent miniature caches in extended DLRU. Compared to Redis, extended DLRU and kRedis reduce the mean access latency by 45.2% and 50.2%, respectively. The time overhead of extended DLRU and kRedis are similar, which are 0.73% and 0.66% of total execution time, respectively, but extended DLRU shows higher space overhead than kRedis, which is 38 times that of kRedis.

It is worth mentioning that in extended DLRU the space and time overhead are directly associated with the number of cache sizes simulated. In order to obtain more accurate K-LRU MRCs, more cache sizes need to be simulated for each tenant. And in a system with a large number of tenants and a greater cache max-memory size, the DLRU simulation overhead for the multi-tenant cache using K-LRU can only be higher. Thus, even though both DLRU and kRedis are efficient for the single-tenant use case, for multi-tenant memory allocation usage, an efficient one-pass MRC modeling algorithm such as EAET and KRR is preferred over interpolation.

7 Related Work

Two classes of studies are related to this work, one is MRC techniques and the other is cache partitioning.

7.1 MRC Techniques

The baseline technique by Mattson *et al.* [41] generates exact MRC for the LRU cache but comes with the cost of extremely large space and time overheads. Many later studies have attempted to reduce the overhead of stack processing by using more compressed stack representation. Olken *et al.* [34] reduced the algorithm complexity down to $O(NlogM)$ by replacing the linear stack structure with the balanced search tree. Scale Tree [46] is a modified version of Olken's stack. 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 which only takes $O(NLog(Log(M)))$ time and $O(LogM)$ space. MIMIR [39] 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. This method takes $O(B)$ time and $O(M)$ space. They demonstrate that MIMIR can generate very accurate MRCs. Counter Stacks [4] 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 was 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 including downsampling and pruning its data matrix as well as replacing the bloom filter-based counter with a low overhead probabilistic cardinality counter. The compressed Counter Stack only requires $O(NLogM)$ time and $O(LogM)$ space to generate accurate MRCs with bounded error. SHARDS [3] further reduces the running time to linear by utilizing a uniform randomized spatial sampling technique, which tracks references that are chosen based on their hash values. This method makes it possible to analyze long traces that were previously difficult due to memory constraints.

More recent advancement in LRU MRC generation uses the metric called reuse time [12], [13], [21]. 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 [13] 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. AET [6], [21], presents a kinetic model for the LRU cache eviction process. This model uses reuse time distribution to compute 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. HOTL [12] shows that the miss ratio of an LRU cache with capacity c can be approximated as the finite difference of average footprint at c, which is equivalent to the probability of reuse time longer than the length of the footprint window.

7.2 Memory Partitioning

Memory allocation is important for the efficiency and performance of in-memory key-value cache systems. Memory pool can be partitioned according to application-related properties, including object size, latency, expiration, frequency, QoS, etc. And the optimization strategy of partitioning is also tailored to satisfy various use case-specified objectives.

LAMA [5] is a locality-aware slab class level memory allocation for Memcached using the footprint theory [12] to model slab class trace locality and construct an MRC for each slab class. LAMA optimizes overall performance for all size classes, either total miss ratio or average response time. It can also be used in QoS-guaranteed applications use cases. Dynacache [38] targets improving the hit rate of web applications, uses a bucketing scheme to estimate item stack distance in trace profiling, and allocates slab memory of Memcached for tenants. Cliffhanger [47] employs a hit rate gradient estimation mechanism using shadow queue structures and incrementally transfers memory resources to the application that would benefit most from those that benefit the least.

Memshare [25] is a DRAM key-value cache memory partitioning system that optimizes the overall hit rate of applications and allows each application to specify its own eviction policy. It extends the Cliffhanger model to track a hit gradient for each application. Memshare abandons slab classes and introduces a segmented in-memory log to store application items. Varied-size items from different applications can be allocated to the same segment and thus memory reallocation can be at the item level. Memshare reserves a specified minimum amount of memory for each application to provide performance guarantees, and the remaining memory is allocated to maximize the hit rate. Both pRedis [40] and mPart [22] adopt AET [21] for online MRC construction and use dynamic programming algorithms guided by tenant MRCs to allocate memory. I-PLRU [48] achieves minimized misses for a multi-flow LRU cache with an insertion-based pooled LRU paradigm. The cache space is pooled to serve multiple data flows but organized as a single list. Each tenant data flow is assigned an insertion position of the list. By configuring the insertion point dynamically, it proves that I-PLRU can reach the same minimum miss ratio of separated LRU caching. Robinhood [23] repurposes existing caches to mitigate backend latency variability. Rather than solely caching popular data, it dynamically reallocates cache resources from "cache-rich" backends, which do not significantly impact request tail latency, to the "cache-poor" backends, thereby increasing hit ratios, reducing backend queries, and easing resource congestion, which all contribute to improved P99 request latency.

8 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 that enables MRC construction for variable object-size K-LRU key-value cache in one pass of the trace. We also propose a fast stack update schemes to further reduce the algorithm's cost. With the support of this random sampling-based LRU model, we present the design and implementation of a lightweight memory partitioning scheme in multi-tenant key-value cache. Besides the capability of capturing trace locality and awareness of reference miss latency, it efficiently explores the potential miss ratio gaps of various sampling sizes in K-LRU. Incorporating spatial sampling, we show that KRR can construct accurate MRC with low space and time overhead. The evaluations over a variety of workloads show that our multi-tenant memory allocation approach achieves better performance than Redis and pRedis. The average access latency is reduced up to 50.2% and 24.8% when compared to Redis

and pRedis, respectively. By adjusting the memory partitioning strategy, we show that the throughput is increased by 262.8% and 61.8% compared to Redis and pRedis, respectively. We also compare kRedis with extended DLRU and discuss the suggested application scenarios for the two. In our future work, we will investigate other random-sampling policies using diverse metrics, such as access frequency and object expiration time, as priority functions, and utilize those in memory management for border usage such as scalable tired memory systems.

ACKNOWLEDGEMENTS

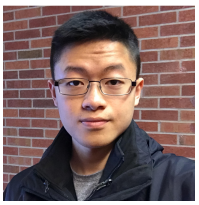
We would like to thank the anonymous reviewers for their constructive comments and suggestions. The research was supported in part by National Science Foundation CSR1618384 and SaTC2225424.

REFERENCES

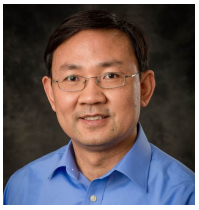
[1] R. Labs, "redis," https://redis.io, Accessed: Sept. 10, 2020.
[2] memcached, "memcached," https://memcached.org, Accessed: Sept. 10, 2020.
[3] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pp. 95–110, USENIX Association, 2015.
[4] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data, "Characterizing storage workloads with counter stacks," in *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pp. 335–349, USENIX Association, 2014.
[5] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "LAMA: Optimized locality-aware memory allocation for key-value cache," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, (Santa Clara, CA), pp. 57–69, USENIX Association, 2015.
[6] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, "Kinetic modeling of data eviction in cache," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016.
[7] M. K. Qureshi and Y. N. Patt, "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, (Washington, DC, USA), pp. 423–432, IEEE Computer Society, 2006.
[8] Y. Xiang, X. Wang, Z. Huang, Z. Wang, Y. Luo, and Z. Wang, "Dcaps: Dynamic cache allocation with partial sharing," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.
[9] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: Virtual memory support for garbage-collected applications," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, (Berkeley, CA, USA), pp. 103–116, USENIX Association, 2006.
[10] J. Chen, L. Chen, S. Wang, G. Zhu, Y. Sun, H. Liu, and F. Li, "Hotring: A hotspot-aware in-memory key-value store," in *18th USENIX Conference on File and Storage Technologies (FAST 20)*, (Santa Clara, CA), pp. 239–252, USENIX Association, Feb. 2020.
[11] E. Berg and E. Hagersten, "Statcache: a probabilistic approach to efficient and accurate data locality analysis," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*, pp. 20–27, IEEE, 2004.
[12] X. Xiang, C. Ding, H. Luo, and B. Bao, "Hotl: a higher order theory of locality," in *ACM SIGARCH Computer Architecture News*, vol. 41, pp. 343–356, ACM, 2013.
[13] D. Eklov and E. Hagersten, "StatStack: Efficient modeling of LRU caches," in *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pp. 55–65, IEEE, 2010.
[14] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 499–511, USENIX Association, 2017.
[15] D. Byrne, N. Onder, and Z. Wang, "Faster slab reassignment in memcached," in *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, (New York, NY, USA), p. 353–362, Association for Computing Machinery, 2019.
[16] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *2nd USENIX Conference on File and Storage Technologies (FAST 03)*, (San Francisco, CA), USENIX Association, Mar. 2003.
[17] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the General Track: 2001 USENIX Annual Technical Conference*, (USA), p. 91–104, USENIX Association, 2001.
[18] L. V. Rodriguez, F. Yusuf, S. Lyons, E. Paz, R. Rangaswami, J. Liu, M. Zhao, and G. Narasimhan, "Learning cache replacement with CACHEUS," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pp. 341–354, USENIX Association, Feb. 2021.
[19] Redis, "Redis replacement policy," https://redis.io/topics/lru-cache, Accessed: Oct. 15, 2019.
[20] N. Beckmann, H. Chen, and A. Cidon, "LHD: Improving cache hit rate by maximizing hit density," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, (Renton, WA), pp. 389–403, USENIX Association, Apr. 2018.
[21] X. Hu, X. Wang, L. Zhou, Y. Luo, Z. Wang, C. Ding, and C. Ye, "Fast miss ratio curve modeling for storage cache," *ACM Trans. Storage*, vol. 14, pp. 12:1–12:34, Apr. 2018.
[22] D. Byrne, N. Onder, and Z. Wang, "mPart: Miss-ratio curve guided partitioning in key-value stores," in *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2018, (New York, NY, USA), p. 84–95, Association for Computing Machinery, 2018.
[23] D. Berger, B. Berg, T. Zhu, M. H, and S. S, RobinHood: Tail Latency Aware Caching-Dynamic Reallocation from Cache-Rich to Cache-Poor," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 195–212, USENIX Association, Nov. 2020.
[24] C. Pan, Y. Luo, X. Wang, and Z. Wang, "pRedis: Penalty and locality aware memory allocation in redis," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, (New York, NY, USA), p. 193–205, Association for Computing Machinery, 2019.
[25] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: a dynamic multi-tenant key-value cache," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 321–334, USENIX Association, 2017.
[26] J. Yang, Y. Wang, and Z. Wang, "Efficient modeling of random sampling-based lru," in *50th International Conference on Parallel Processing*, ICPP 2021, (New York, NY, USA), Association for Computing Machinery, 2021.
[27] Y. Wang, J. Yang, and Z. Wang, "Dynamically configuring lru replacement policy in redis," in *The International Symposium on Memory Systems*, MEMSYS 2020, (New York, NY, USA), p. 272–280, Association for Computing Machinery, 2020.
[28] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache Modeling and Optimization using Miniature Simulations," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, (Santa Clara, CA), pp. 487–498, USENIX Association, 2017.
[29] SNIA, "Msr cambridge traces," http://iotta.snia.org/traces/388, Accessed: March 15, 2020.
[30] D. Narayanan, A. Donnelly, and A. Rowstron, "Write Off-Loading: Practical Power Management for Enterprise Storage," in *6th USENIX Conference on File and Storage Technologies*, FAST 08 2008, (San Jose, CA, USA), p. 253–267, USENIX Association, 2008.
[31] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.
[32] J. Yang, Y. Yue, and K. V. Rashmi, "A large scale analysis of hundreds of in-memory cache clusters at twitter," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 191–208, USENIX Association, Nov. 2020.
[33] Twitter, "Twitter Cache Trace,"https://github.com/twitter/cache-trace, Accessed: Dec. 2020.
[34] F. Olken, "Efficient methods for calculating the success function of fixed-space replacement policies," tech. rep., Lawrence Berkeley Lab., CA (USA), 1981.
[35] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction

(rrip)," *SIGARCH Comput. Archit. News*, vol. 38, p. 60–71, June 2010.
[36] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang, "Optimizing locality-aware memory management of key-value caches," *IEEE Transactions on Computers*, vol. 66, pp. 862–875, May 2017.
[37] X. Xiang, B. Bao, C. Ding, and Y. Gao, "Linear-time modeling of program working set in shared cache," in *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pp. 350–360, IEEE, 2011.
[38] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *Proceedings of the 7th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'15, (Berkeley, CA, USA), pp. 19–19, USENIX Association, 2015.
[39] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, (New York, NY, USA), pp. 28:1–28:14, ACM, 2014.
[40] C. Pan, X. Wang, Y. Luo, and Z. Wang, "Penalty- and locality-aware memory allocation in redis using enhanced aet," *ACM Trans. Storage*, vol. 17, May 2021.
[41] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM System Journal*, vol. 9, no. 2, pp. 78–117, 1970.
[42] G. Bilardi, K. Ekanadham, and P. Pattnaik, "Efficient stack distance computation for priority replacement policies," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, (New York, NY, USA), Association for Computing Machinery, 2011.
[43] B. Berg, D. S. Berger, S. McAllister, I. Grosof, S. Gunasekar, J. Lu, M. Uhlar, J. Carrig, N. Beckmann, M. Harchol-Balter, and G. R. Ganger, "The cachelib caching engine: Design and experiences at scale," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 753–768, USENIX Association, Nov. 2020.
[44] C. Pan, X. Hu, L. Zhou, Y. Luo, X. Wang, and Z. Wang, "Pace: Penalty aware cache modeling with enhanced aet," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.
[45] Redis, "Hiredis,"https://github.com/redis/hiredis, Accessed: Sept. 10, 2021.
[46] Y. Zhong, X. Shen, and C. Ding, "Program locality analysis using reuse distance," *ACM Trans. Program. Lang. Syst. (TOPLAS '09)*, vol. 31, pp. 20:1–20:39, Aug. 2009.
[47] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, (Santa Clara, CA), pp. 379–392, USENIX Association, 2016.
[48] G. Quan, J. Tan, A. Eryilmaz, and N. Shroff, "A new flexible multi-flow lru cache management paradigm for minimizing misses," *Proc. ACM Meas. Anal. Comput. Syst.*, vol. 3, Jun 2019.

Yuchen Wang received his graduate degrees from Huazhong University of Science and Technology, China. He is currently working toward the Ph.D. degree in the Department of Computer Science, Michigan Technological University, USA. His research interests include memory system optimization and distributed systems.



Junyao Yang received his B.S. and M.S. from Michigan Technological University. He is currently working towards the Ph.D. degree in the Computer Science Department at Michigan Technological University. His research interests include storage systems, software cache modeling and optimization.



Zhenlin Wang received his Ph.D. degree in Computer Science in 2004 from the University of Massachusetts, Amherst. He is now a full professor of the Department of Computer Science, Michigan Technological University, USA. His research interests are broadly in the areas of compilers, operating systems and computer architecture with a focus on memory system optimization and system virtualization.