Dynamically Configuring LRU Replacement Policy in Redis

Yuchen Wang
yuchenwa@mtu.edu
Michigan Technological University
Houghton, Michigan

Junyao Yang
junyaoy@mtu.edu
Michigan Technological University
Houghton, Michigan

Zhenlin Wang
zlwang@mtu.edu
Michigan Technological University
Houghton, Michigan

Abstract

To reduce the latency of accessing backend servers, today's web services usually adopt in-memory key-value stores in the front end which cache the frequently accessed objects. Memcached and Redis are two most popular key-value cache systems. Due to the limited size of memory, an in-memory key-value store needs to be configured with a fixed amount of memory, i.e., cache size, and cache replacement is unavoidable when the footprint of accessed objects is larger than the cache size. Memcached implements the least recently used (LRU) policy. Redis adopts an approximated LRU policy to avoid maintaining LRU list structures. On a replacement, Redis samples pre-configured K keys, adds them to the eviction pool, and then chooses the LRU key from the eviction pool for eviction. We name this policy *approx-K-LRU*. We find that approx-K-LRU behaves close to LRU when K is large. However, different Ks can yield different miss ratios. On the other hand, the sampling and replacement decision itself results in an overhead that is related to K. This paper proposes DLRU (*Dynamic LRU*), which explores this configurable parameter and dynamically sets K. DLRU utilizes a low-overhead miniature cache simulator to predict miss ratios of different Ks and adopts a cost model to estimate the performance trade-offs. Our experimental results show that DLRU is able to improve Redis throughput over the recommended, default approx-5-LRU by up to 32.5% for a set of storage traces.

CCS Concepts: • **Information systems** → **Application servers**; **Cloud based storage**.

Keywords: Redis, Cache Replacement, Memory Allocation, In-Memory Key-Value Stores, Memory Caches, LRU

MEMSYS 2020, September 28-October 1, 2020, Washington, DC, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8899-3/20/09…$15.00
https://doi.org/10.1145/3422575.3422799

ACM Reference Format:
Yuchen Wang, Junyao Yang, and Zhenlin Wang. 2020. Dynamically Configuring LRU Replacement Policy in Redis. In *The International Symposium on Memory Systems (MEMSYS 2020), September 28-October 1, 2020, Washington, DC, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3422575.3422799

1 Introduction

In today's multi-level storage architectures, in-memory key-value cache plays an important role to ensure low-latency system performance. Retrieving objects from a key-value cache system deployed in memory in a front-end server is much faster than from a remote backend server. Redis is one of the well-developed in-memory cache systems to reduce response time and alleviate load pressure on databases. It has been widely deployed by many companies and in countless mission-critical production environments, including Twitter, GitHub, Weibo, StackOverflow, Flickr, Amazon and many other popular web services [2, 19, 21].

An important feature of Redis is that it supports not only strings, but also typed objects such as sets, hashes, lists, bitmaps, geospatial data etc. [18], making it the first choice in the cases that different types of data must be cached . When Redis reaches its pre-configured memory usage limit, in order to store newly incoming data it must evict old data to make available memory space. This process is called data eviction, and the strategy of deciding which data to be evicted is known as the eviction policy or replacement policy. Redis supports various eviction policies including Least Recently Used (LRU), Least Frequently Used (LFU), Random Eviction, and Shorter Time-to-Live (TTL) [20]. Among those policies, LRU replacement is the most commonly chosen policy in industry for both software and hardware caches.

Research of hotspot issue shows that accesses in real-word commercial key-value stores follow the power-law distribution where the popular keys dominate the accesses [6]. LAMA's evaluation on the Facebook ETC workload also shows its high data locality [9]. LRU is therefore the good choice as it can exploit the locality well. However, 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 [3], 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, it

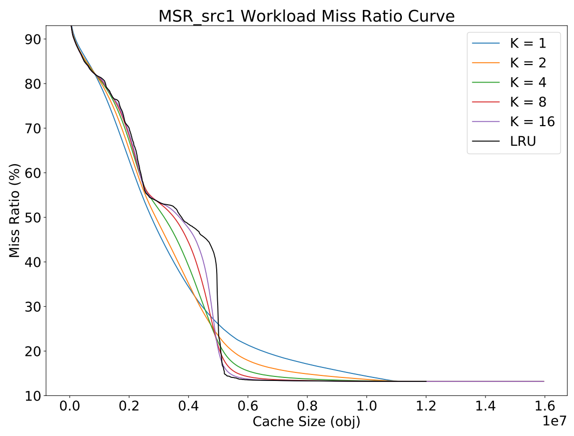


Figure 1. MRCs of src1 under LRU and different Ks

must be locked to facilitate the update of the corresponding LRU priority, resulting in extra performance degradation [5]. Memcached, another popular key-value store, only maintains the LRU structure at the slab class level [14].

In order to save memory and improve performance, Redis discards the ordered list structure design and applies an approximated LRU policy [20] that only needs to keep track of the access time of each object. Redis picks the candidate with the oldest access time for eviction from an eviction pool consisting of only a small number of keys. Each time when an eviction is needed, according to a specified sampling configuration, K keys are randomly sampled from all keys in the memory and added into 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. We name this eviction policy, *approx-K-LRU*. The default setting of Redis is $K = 5$ meaning each time five randomly sampled keys are added to the pool for eviction decision. K is configurable but is fixed across Redis execution for the current design.

We run a collection of real-word enterprise server traces from Microsoft Research Cambridge [1] on Redis to plot the miss ratio curves (MRCs), which show the miss ratios against the maximum memory size (cache size) of Redis. Through experiments, we observe that the MRCs of approx-16-LRU can closely approximate those of the real LRU. The difference between the two are typically minimal enough to ignore. A larger K normally indicates a closer behavior of eviction to a real LRU policy, since the more keys are sampled to add into the eviction pool, the more likely the evicted object is near the LRU side of a real LRU list.

Figure 1 shows the MRCs of five different sample sizes for trace `src1` where the cache size is represented as the number of objects. We can clearly observe that the sample size K could impose large impacts on miss ratios. When the cache size is less than 0.5 * 1e7, the miss ratio of approx-1-LRU is almost always lower than other settings of K, which means the random replacement policy can perform better than LRU. However if the cache size is greater than this point, the MRC of LRU apparently indicates that the real LRU eviction policy is the best choice if we only consider the impact of miss ratio. The maximum difference of miss ratio under various K settings at a fixed cache size could be more than 10%. Research has shown that the overall performance of cache systems are highly determined by its miss ratio, even a slight reduction of it could introduce a significant improvement in performance [4, 7].

This paper proposes *DLRU: Dynamic LRU*, which explores the configuration of K in approx-K-LRU, in order to improve the overall system performance. DLRU reconfigures K along with Redis execution. We adopt a scaled-down cache simulator to track the miss ratios of different Ks on the fly with a low overhead [22]. We develop a cost model to balance between the benefit of low miss ratio and the overhead of random selection and sorting of approx-K-LRU. Our experiment results show that DLRU can always match the performance of the best K, and improve the overall Redis throughput over the default approx-5-LRU by up to 32.5%.

2 Motivation and Background

This section further motivates our work by illustrating approx-K-LRU eviction through a trace example. Then we describe Miniature Cache [22], a low-overhead cache simulator. DLRU uses miniature cache to emulate approx-K-LRU and estimate the miss ratio.

2.1 Impacts of Sample Size K

We use a simple trace example to illustrate the impact of sample size K over miss ratio by choosing three settings of K, where $K = 16$ simulates the real LRU, $K = 2$ represents a middle ground, and $K = 1$ means random eviction. Tables 1 to 3 show the cache content and eviction selection for approx-16-LRU, approx-2-LRU, and approx-1-LRU, respectively. Assume that cache capacity is 5. An evicted key is the one with largest access time chosen from K sampled keys, which are sorted based on their recent access time from top to bottom with the most recent item on top in the tables. Under $K = 16$, every access is a miss, miss ratio is 100%. Under approx-2-LRU, there are 9 misses and the miss ratio is 75%. Under random eviction, i.e., approx-1-LRU, there are 7 misses and the miss ratio has decreased to 58%.

This example shows that the trace pattern, cache size, and sample size K of approx-K-LRU all have impacts on the miss ratio of a key-value cache. As pointed by Jaleel et al [12], 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 an access and its next reuse is the number of distinct accesses in between. In the example trace,

Trace	a	b	c	d	e	f	a	b	c	d	e	f
Cached Keys	a	ab	abc	abcd	abcde	bcdef	cdefa	defab	efabc	fabcd	abcde	bcdef
Evicted Key						a	b	c	d	e	f	a
Sampled Keys						e	f	a	b	c	d	e
						d	e	f	a	b	c	d
						c	d	e	f	a	b	c
						b	c	d	e	f	a	b
						a	b	c	d	e	f	a
Hit	0	0	0	0	0	0	0	0	0	0	0	0

Table 1. Redis replacement: approx-16-LRU

Trace	a	b	c	d	e	f	a	b	c	d	e	f
Cached Keys	a	ab	abc	abcd	abcde	acdef	cdefa	cefab	efabc	eabcd	abcde	acdef
Evicted Key						b		d		f		b
Sampled Keys						e		a		c		e
						b		d		f		b
Hit	0	0	0	0	0	0	1	0	1	0	1	0

Table 2. Redis replacement: approx-2-LRU

Trace	a	b	c	d	e	f	a	b	c	d	e	f
Cached Keys	a	ab	abc	abcd	abcde	abdef	bdefa	defab	defac	efacd	facde	acdef
Evicted Key						c			b			
Sampled Keys						c			b			
Hit	0	0	0	0	0	0	1	1	0	1	1	1

Table 3. Redis replacement: approx-1-LRU

the length of the reuse intervals for all reuses is 6, which is bigger than the cache size 5. In this case, approx-16-LRU, which behaves close to the real LRU, fails to generate any hits. On the other hand, random eviction leads to most hits. In reality, we can observe a mix of short and long reuse intervals in different phases. Based on this observation, we propose an approach to choosing K dynamically. However, miss ratio cannot be the only metric to make the decision. A larger K implies more overhead in the eviction process than that of a smaller K. We will describe a cost model in Section 3.5.

2.2 Miniature Simulation

In order to make selection of sample size K, the miss ratio of the corresponding K must be predicted in real time. We adopt a lightweight scaled-down approximation technique, the Miniature Cache Simulator, to reduce overhead of capturing trace pattern and miss ratio tracking [22]. The miniature cache proposed by Waldspurger et al. simulates the actual cache by scaling down, by several orders of magnitude, both the original accesses and the cache size. It can accurately model the behavior of the original cache with any given eviction policy.

During the processing of an online trace, all references are hashed and only when the hash value of a key is less than a threshold, that key of reference is sampled and stored (cached) into a hash table. Let the T and P be threshold and modulus. The sampling condition for any referenced key L is

$$hash(L) \mod P < T \quad (1)$$

The miniature cache sampling rate is thus $R = T/P$. This ensures all requests to the same key will be sampled. The small and spatially hashed samples of the requests show a statistical similarity against the whole references. Miss ratio then can be extracted from the miniature cache by counting the number of misses against the total sampled references. Experiments show that small values of $R = 0.01$ or even $R = 0.001$ can yield very accurate results. Such a low sampling rate implies low time and space overhead even for a long execution.

3 Design and Implementation

In this section, we describe the design and implementation of DLRU in detail. Our implementation is above Redis eviction which is described in Section 3.1. We then present an overview of DLRU design in Section 3.2. Section 3.3 describes the implementation of the miniature caches for miss ratio prediction. Section 3.4 presents a design to measure miss latency and eviction process overhead, which are taken into consideration in the cost model in Section 3.5. The cost model determines the choice of K for DLRU.

3.1 Redis Eviction Process

We use approx-5-LRU as an example to explain the Redis workflow for evictions. First Redis initializes it's server according to configurations including $K = 5$. Assume a client sends a command, GET key. Redis looks up the command handling table to find which function to use to handle this request. If it is an access that hits cache, Redis returns the value to the client. If it is a miss, Redis returns a miss flag to the client. In our experiments, once a client receives a miss notice, it sends a SET request with the key and value. For any set request, if the key is new for Redis, it conducts a memory eviction check. If the max-memory limit is reached, the eviction process will be invoked. Five (K) keys are randomly sampled from the memory key space and merged into the current eviction pool. Then the one with the oldest last-access-time will be evicted. The size of the eviction pool is fixed and configurable. We use the default configuration of 16 for all our evaluation. When the sampled five keys are merged into the eviction pool, the youngest five keys are dropped. Redis repeatedly checks if the currently used memory is under the max-memory limit. If not, a new round of eviction is invoked. Once Redis has spared enough memory space, the new key-value pair is set into memory.

3.2 DLRU Overview

As discussed in Section 3.1, Redis sets up K during initialization and K is fixed unless a client manually switches it. The eviction process does not require that K be fixed. So the basic idea of DLRU is simple: reset K automatically on the fly. As shown in Figure 2, we simulate approx-K-LRU under various K. We dedicate one miniature cache for each K. We use a penalty cost model to pick one that minimizes overall time latency and reconfigure Redis in real time. The implementation consists of two parts residing in server initialization and command dispatcher, respectively. In server initialization, we set an interval size measured as the number of GET requests. Later, for every interval, DLRU will decide if a new K needs to be set. We also initialize the miniature caches in this stage. In the command dispatcher, once a GET key command is detected, DLRU determines whether to sample such key. If a key is sampled, it is fed into every miniature cache. After calculating and comparing overall miss penalty for each K, the Redis `server.maxmemory_samples` parameter is set to be the optimal K with the least predicted penalty.

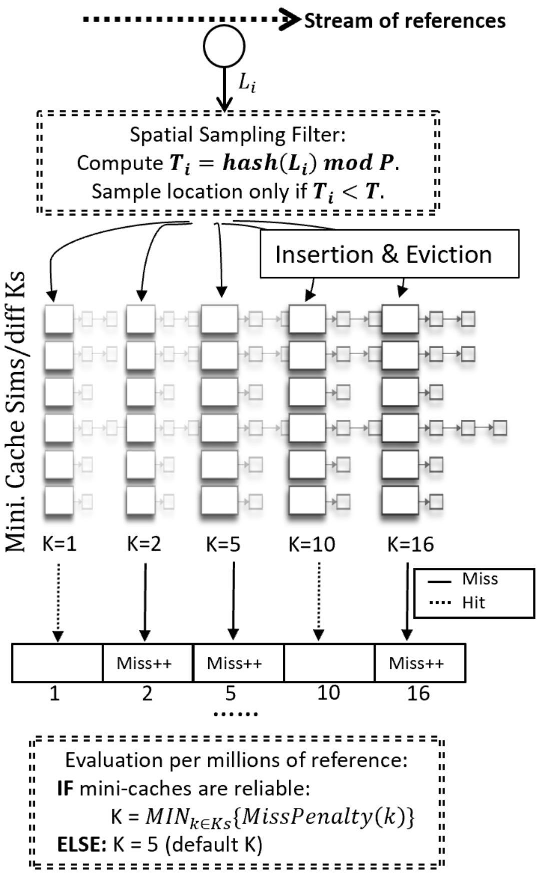

Stream of references
L_i
Spatial Sampling Filter:
Compute $T_i = hash(L_i) \; mod \; P$.
Sample location only if $T_i < T$.
Insertion & Eviction
Mini. Cache Sims/diff Ks
K=1
K=2
K=5
K=10
K=16
Miss
Hit
Miss++
Miss++
Miss++
1
2
5
10
16
Evaluation per millions of reference:
IF mini-caches are reliable:
K = $MIN_{k \in Ks}\{MissPenalty(k)\}$
ELSE: K = 5 (default K)

Figure 2. DLRU overview

3.3 Miniature Cache Simulation

We place a filter in Redis to identify keys that satisfy condition in Eq. 1. Modulus P is set to a power of two, and threshold T is fixed according to a given miniature cache sampling rate R as $T = P * R$. In a periodic interval window (default is 5 million GET requests), a randomly sampled small subset of references are selected to feed into the miniature caches.

As discussed in Section 1, $K = 16$ and $K = 1$ represent real LRU and random eviction, respectively. We add three other settings in between where $K = 2$, $K = 5$, and $K = 10$. Five independent approx-K-LRU miniature caches, corresponding to K = 1, 2, 5, 10, 16, are fed with this subset of keys at the same time. Although more miniature caches of different Ks between 1 and 16 could be evaluated, the performance gap between a small interval of K-settings is not significant. In addition, more miniature caches will also introduce more simulation overhead.

Each miniature cache first looks up the sampled keys in their own key space, which is maintained in their own hash table. If the key does not exist, a miss occurs and the key is cached. A cache replacement using the corresponding

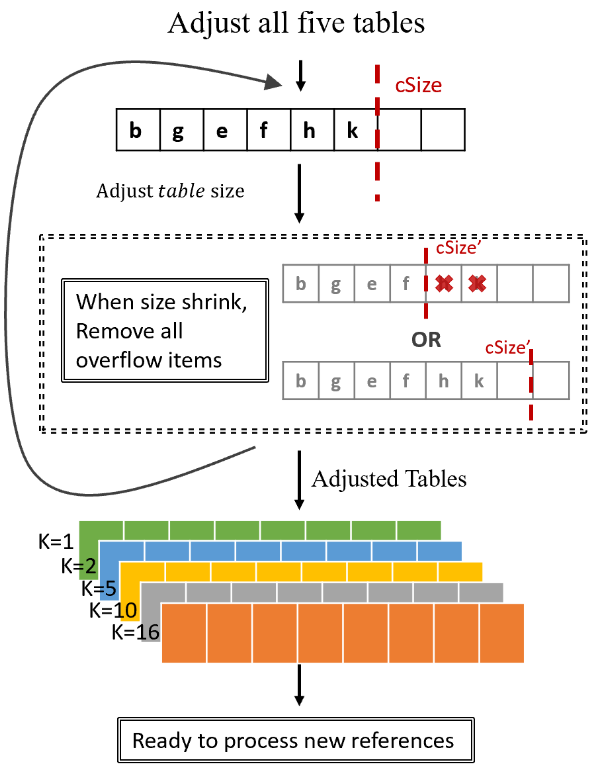


Figure 3. Hash table resizing

approx-K-LRU policy is invoked if the miniature cache is full. To determine if a miniature cache is full, the current used cache size is compared with the maximum cache size. The current used cache size is simply the number of items stored in the miniature cache. The maximum cache size, in terms of the number of objects, is computed using Eq. 2 below, where the average item size is extracted directly from Redis statistic. As the average item size can change across intervals, the *Max cSize* should adjust accordingly. As shown in Figure 3, all miniature caches are required to adjust its size after every five millions of requests. When the *Max cSize* shrinks, we simply remove all overflow items in the miniature cache. Note that since the position of item in the miniature cache is randomly determined, the removals of n items from the tail of miniature cache are statistically equivalent to randomly removing n items.

$$\text{Max cSize} = \text{server.maxmemory} * R/\text{average item size} \quad (2)$$

Ideally, miniature caches should simulate approx-K-LRU accurately. However, if during an interval, not enough distinct references are cached, the accuracy is not guaranteed [22]. We observe that when the actual number of sampled distinct references in an five million request interval is greater than 256, miniature simulation can deliver acceptable accuracy, which is also observed in the original work by Waldspurger et al [23]. In our implementation, we set R to 1/200, i.e., one every 200 requests is sampled. This sampling rate works well for most cases. If in any evaluation interval, less than 256 distinct references are collected, we consider that the miniature cache is too small to predict reliable miss rate. In this case, the default $K = 5$ is configured for the following interval. Note that 256 is a very small number compared to the total of five million requests. Only on a few occasions, DLRU will need to go back the default.

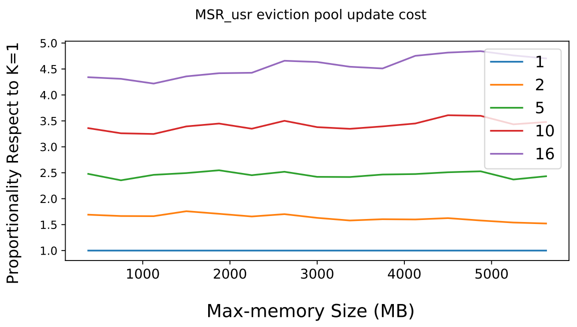


Figure 4. Overhead measurement

3.4 Miss Latency and Eviction Process Overhead

To estimate the performance impact of misses, we need to know the miss 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 client. We measure the miss latency on the fly for each interval of 5 million references. We track the miss penalties in the previous interval and use their mean as the miss latency for the DLRU decision in the next interval.

The cache eviction overhead comes from sampling and access-time comparison. The first one is the operation of randomly sampling the required number of K keys from all that in memory. The other is the operation of merging with the eviction pool and finding a key with the largest last access time for eviction. It is challenging to actually measure this overhead for all approx-K-LRU settings in real time as in any time interval, only one setting of K can be measured.

In our experiments, we observe a proportional relationship of such overhead between different settings of K. Let OH_K be the mean sampling and comparison overhead for approx-K-LRU. Figure 4 shows OH_K for $K = 1, 2, 5, 10, 16$ for `usr` on a fixed-K-configuration Redis server. The range of the each normalized curve is relatively small. In other words, if we measure the OH_K at a time window, we can estimate other $OH_K^{\prime}s$ based on the pre-measured proportionality ratios.

3.5 DLRU Cost Model

In order to deal with access pattern changes, we divide the reference stream into fixed-size intervals according to the number of GET requests. In our experiments, the default interval size is set to 5 million GET accesses. The cost model in this section estimates the overall miss penalty for approx-K-LRU in the current interval and use it to guide the choice of K for the next interval.

Let p be the average miss latency, OH_K be eviction overhead and M_K be the miss count gathered in the past interval using the miniature model. We estimate P_K the overall miss penalty for approx-K-LRU as follows.

$$P_K = M_K * (p + OH_K) \tag{3}$$

Our goal is to choose a K with the minimal P_K. Then Redis server is reconfigured with the optimal K for the following interval. Normally, p is orders of magnitude greater than OH_K, so the impact of the latter over the overall miss penalty is trivial. Selection of K is dominated by the miss counts predicted by the miniature caches. Our scheme will choose the K with the smallest miss ratio. However, we observe that, if the miss counts are very close to each other for different Ks in an interval, OH_K can become a deciding factor for overall miss penalty. In this case, our scheme will prefer a smaller K.

4 Evaluation

In order to evaluate the effectiveness of DLRU, we first give a brief description of experimental setup. Second, we evaluate the accuracy of the predicted miss ratio. Third, we compare the performance difference between Redis with default K and Redis+DLRU. Finally, we discuss both time and space overhead of our selection scheme.

4.1 Experimental Setup

4.1.1 System Configuration. We use two separate machines for evaluation. Machine A 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 16.04.6 LTS with Linux kernel 4.4.0. Machine B is configured with Intel(R) Xeon(R) Gold 5118 2.30GHz processor with 33 MB shared LLC and 192 GB of memory and the operating system is Fedora 31 with Linux kernel 5.6.13. All major evaluations are done on machine A, Machine B is only used in Section 4.4. We have implemented DLRU on top of Redis-4.0 [17] with the default Jemalloc allocator, and use *mutilate* [15] for request stream generation.

Initially mutilate converts references in a workload to Redis GET commands, when Redis returns a miss, Mutilate will immediately follow a SET command. There is no back-end database in our setup, all KV pairs are generated from the mutilate client on the fly. With such setup, the miss latency is simply the total setback time between mutilate and Redis. Additionally, both Redis and mutilate are running on localhost. It yields relatively low access latency when compared to more typical cases where clients are run on a remote site. In a real system, the miss latency will be much higher. DLRU will still function as it measure the miss latency on the fly. With a higher miss latency, DLRU can only perform better as the overall miss penalty is higher.

4.1.2 Workloads. We use the MSR Cambridge storage workloads and its variants in our evaluation [1]. The original MSR suite contains traces from 13 different enterprise data center servers. It covers a variety of access patterns, which is sufficient for us to evaluate the effectiveness of DLRU. We first evaluate the MSR traces under simplified conditions with uniform object size, where the object size of each key value pair is 200 bytes. Next, we use the MSR traces' original object size to show that DLRU also improves the performance of Redis under general circumstances. The MSR suite contains a couple small traces which only take Redis roughly 10 minutes to process entire request streams. In order to better visualize the improvement from DLRU, we repeatedly concatenate the same trace to coin a roughly one-hour long request stream. For notation purpose, as an example, `src2_10` is generated by concatenating MSR's `src2` trace 10 times. Lastly, in Section 4.3.3, we merge multiple MSR traces with different access patterns to demonstrate how DLRU selects an optimal K when access pattern changes.

4.2 Miss ratio

For DLRU to make meaningful decisions, the five miniature caches must correctly simulate Redis replacement patterns under different Ks. We compare actual Redis miss ratio for every 5 million requests with predicted miss ratio yielded by the miniature cache with the corresponding K. Figure 5 shows the miss ratios over time for all MSR traces. To quantify the accuracy of predicted miss ratio, we follow the error metric used in [23], the mean absolute error (MAE). We calculate the MAE for each trace in Figure 5, and the average MAE across all traces is 0.031. We notice that there is an obvious vertical shift between predicted and actual miss ratio for the workloads with relative small working set size such as `stg_16`. In practice, we find that the shift is consistent for all Ks and it is not a problem of DLRU decision. We attribute the shift to the bias introduced by spatial sampling of miniature modeling. Since all five miniature caches use the same subset of keys from spatial sampling, they are likely to suffer from the same relative vertical shift.

In Figure 6, we use a synthetic workload that contains two separate phases. One phase is designed with poor temporal reuse, where random evictions are more preferable, and the other phase is designed with high temporal reuse, where evicting the LRU objects are more preferable. Figure 6 contains miss ratio predicted by DLRU and Redis miss ratio. We also plot miss ratios for both miniature cache with K =

1 and 16 to illustrate that DLRU always selects the optimal K over time. Note that in the initial phase where the miss ratio of approx-1-LRU and approx-16-LRU are roughly the same, DLRU chooses $K = 1$ (the choice of DLRU is shown in the square boxes). This is the case when the eviction overhead decides the K selection as a smaller K implies a lower overhead (OH_K in Eq. 3).

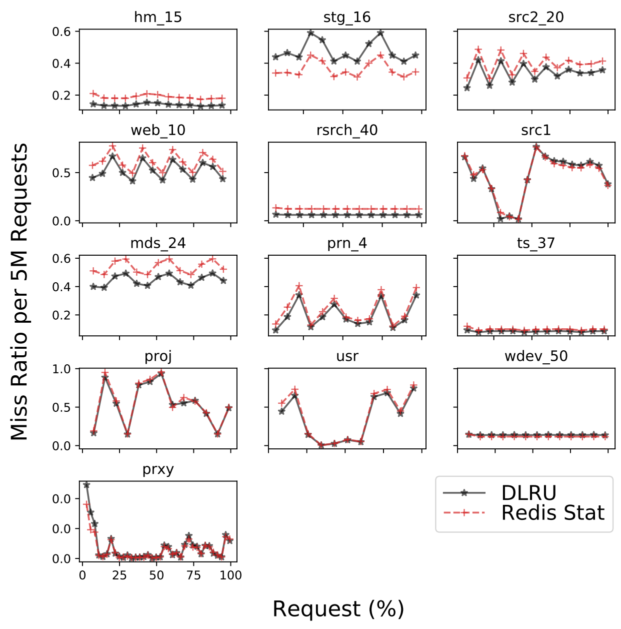


Figure 5. Miss ratio prediction accuracy

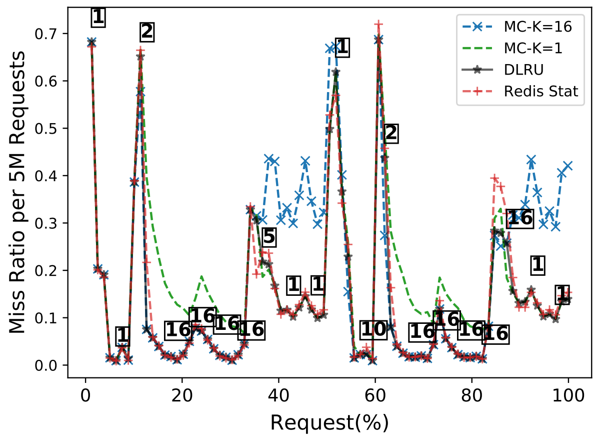


Figure 6. Miss ratio prediction for a phase changing workload

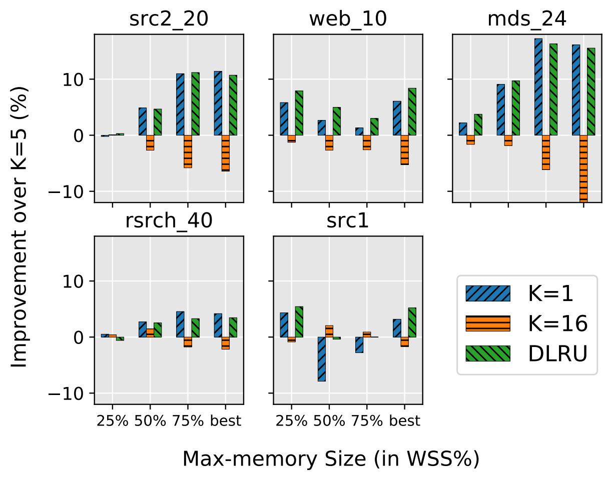
4.3 Overall Throughput

In order to measure the performance gain of our model, we employ throughput as the evaluation metric. Since all workloads we use are fixed-length traces, throughput is the ratio of the total number of requests to the overall execution time. In this section we compare DLRU, approx-1-LRU and approx-16-LRU, with Redis default sample size $K = 5$ (approx-5-LRU). In practice, the approx-16-LRU behaves almost identical to true LRU, and the approx-1-LRU is basically the random replacement policy. We will demonstrate the benefit of DLRU that exploits the access pattern of the current request stream on the fly.

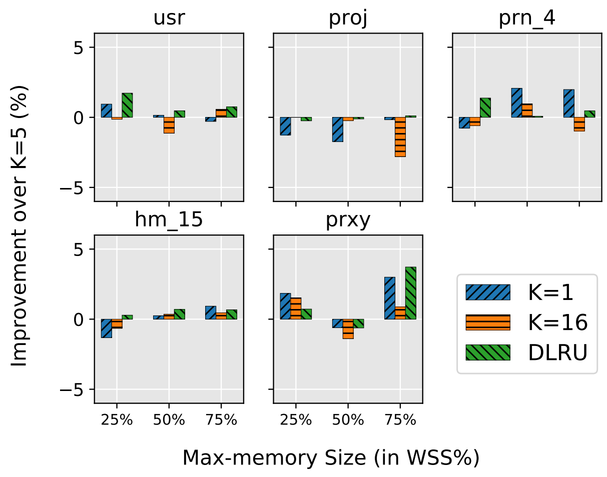
4.3.1 Uniformly-Sized MSR Workloads. In this set of experiments, we set the item size uniformly to 200 bytes for all MSR workloads. We divide the 13 MSR workloads into two separate sets, A and B. Set A includes those MSR workloads that have notable difference in terms of miss ratio under various Ks (1, 2, 5, 10, 16). Many workloads in set A consist of long-stream repeated patterns which are in favor of random replacement when Redis' max-memory is smaller than their working set sizes (WSSs). Set B consists of the MSR workloads that have relatively small difference in terms of miss ratio under various K. Figure 7a and Figure 7b shows results from 5 representative MSR workloads in set A and B, respectively. To evaluate the performance of DLRU under different Redis' `max-memory`, the Redis `max-memory` is set to 25%, 50%, and 75% of the working set size of the evaluated workload. The "best" in Figure 7a is the memory size where there is the largest gap in miss ratio between $K = 1$ and $K = 16$.

In set A, compared to default sample size K=5, DLRU increases throughput by as much as 16.3%. DLRU matches or outperforms approx-5-LRU in all benchmarks and all max-memory settings. It is worth noting that when max-memory is set to 25% of WSS, the `src1` workload shows favor to random replacement ($K = 1$). We see 4% and 5.5% improvement for $K = 1$ and DLRU, respectively. Then we see an 8% degradation for random replacement when max-memory is set to 50% of WSS. The MRC of `src1` (see Figure 1) reflects that about 40% of all items in the workload have high access frequency. When the max-memory is set to 50% of WSS, Redis is able to keep all hot items, random replacement is no longer the favorite choice. DLRU's auto selection of K is able to perform the best in both cases.

In set B, as shown in Figure 7b, the largest improvement by DLRU is 3.7% from `prxy`, which is modest compared to the workloads in set A. Set B consists of workloads that are insensitive to change in K, i.e., all workloads performs mostly same under random replacement or LRU replacement, which results in limited room for improvement under DLRU. But on the upside, we still see that DLRU increases throughput of all workloads by 1%, on average, compared to approx-5-LRU. In set B, both random replacement ($K = 1$) and approx-16-LRU



(a) Set A (best: memory size that yields the largest difference in term of miss ratio)



(b) Set B

Figure 7. Throughput improvement with respect to approx-5-LRU for uniform item size

($K = 16$) perform nearly identical to default $K = 5$, with a difference of 0.3% and -0.2%, on average, respectively.

4.3.2 Non-Uniformly-Sized MSR Workloads. Next we evaluate the performance of DLRU under non-uniformly-sized items. Figure 8 shows the results from two representative MSR workloads, where the size of each item is directly adopted from the original MSR traces. As the increased item size increases the miss penalty, we observe better improvement in some workloads. For `stg_16`, DLRU increases the throughput by 32.5% compared to default $K = 5$ at the memory size of 75% WSS. In `stg_16`, the item size distributions are relatively stable across DLRU intervals. Our model, which uses average item size for cache simulation, works well. However, for `rsrch_40`, the average item size fluctuates across request stream, which hurts the miniature cache accuracy. Despite of this drawback, DLRU still shows the best performance at 25% and 75% WSSs, while there is a slight degradation compared to the default at 50% WSS.

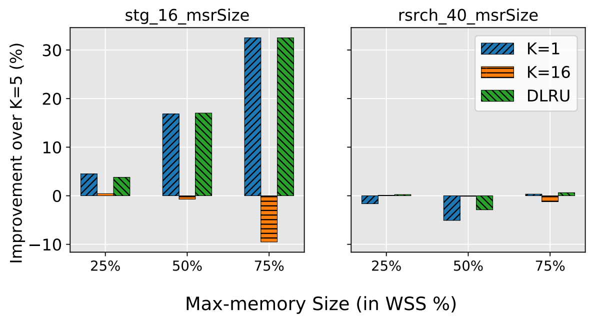


Figure 8. Throughput improvement with respect to approx-5-LRU for nonuniform item size

4.3.3 Synthetic Two-Phase Workload. We evaluate the performance of the two-phase workload discussed in Section 4.2. Note that the workload consist of phases favoring over random replacement and phases favoring over LRU. As shown in Figure 6, a static choice of K would fail to make the best out of both phases. Figure 9 shows the improvement from DLRU when Redis' maxmemory is set to 30% of the WSS. The overall throughput is improved by 6.4% when compare against default $K = 5$.

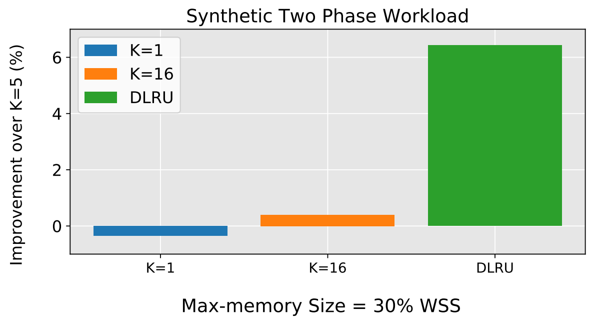


Figure 9. DLRU improvement on a two-phase workload

4.4 Sensitivity

As mentioned in Section 3.4, we observe a proportional relationship in cost of updating the eviction pool under different settings of K. To verify that such observation is consistent over different machines, we collect the cost on both machine A and machine B (See Section 4.1.1) with Redis set to various max-memory size (10MB - 9GB) . Table 4 shows the mean constant of proportion ratios with respect to $K = 1$ and their standard deviation over various max-memory sizes. The standard deviation is low. The results from both machine A and machine B agree with our observation: The costs of eviction under different settings of K are relatively proportional.

Machine A		
K	Ratio	SD
1	1.00	0.00
2	1.64	0.05
5	2.37	0.06
10	3.18	0.13
16	4.31	0.18

Machine B		
K	Ratio	SD
1	1.00	0.00
2	1.64	0.09
5	2.47	0.14
10	3.37	0.28
16	4.40	0.43

Table 4. Ratio of eviction process cost in Redis under different settings of K

4.5 DLRU Overhead

4.5.1 Space Overhead. In our implementation, the space overhead is dominated by the five hash tables, which are used to simulate cache behavior under various K. When applying a fixed rate version of the miniature cache, the size of the hash table will depend on both sampling rate R and the average item size. Each item in the hash table, including auxiliary fields such as hash handle, consumes 136 bytes. We can estimate the percentage of memory overhead relative to overall allocated Redis memory as following: $136\ bytes\ *\ 5\ Tables\ *\ R\ /\ average\ size\ of\ KV\ pair$. As an example, the `stg_16` trace contains 1.6 million unique Key-Value pairs, the average size of each KV pair is 70KB and we set $R = 1/200$. In this case the total additional space overhead introduced by DLRU is about 0.005% of overall allocated Redis Memory.

4.5.2 Time Overhead. The time overhead of DLRU mostly comes from simulating miniature caches under various K. The miniature cache technique helps reduces time overhead drastically. We only sample one request for roughly every $1/R$ requests (one in every 200 in our evaluation). For `stg` with average key-value pair size of 70KB, we observe that the time overhead of DLRU is only 0.027% of total execution time, which is insignificant compared to the potential gain from DLRU. The time overhead for other workloads are similarly low.

5 Related Work

Two classes of studies are related to this work, one is cache replacement algorithm and the other cache modeling. Although LRU and its approximations have become the *de facto* standard replacement policy for both software caches and hardware caches, decades of efforts to improve LRU have never stopped. Jaleel et al's RRIP outperforms LRU by predicting re-reference (reuse) interval and exploiting long intervals that can occur in certain representative benchmarks [12]. Hyperbolic caching proposes a caching algorithm that combines both recency and frequency for replacement decision [3]. The now classic multi-queue algorithm handles recency and frequency by structuring multiple LRU queues based on their frequencies [26]. Even Redis itself implements other replacement policies in addition to approx-K-LRU. Our research focuses on dynamic selection of replacement policy. The approach can be applied to include other policies into the selection.

LRU is a focus of the research on cache modeling too. The nominal work by Mattson *et al* shows that an LRU cache can be modeled as a stack and the miss ratio curve (MRC) can be constructed through one pass of the input trace [13]. Their algorithm measures the stack distance, which is often called reuse distance, for each reference. The reuse distance distribution can be transformed to an MRC. The recent advancements include the footprint theory [25], StatStack [8], SHARDS [23], CounterStacks [24], AET [10, 11], and EAET [16]. CounterStacks improve over Mattson's algorithm by approximating MRC with a novel data structure to compress the reuse distances. SHARDS on the other hand scales down reuse distance measurement though spatial sampling. The footprint theory, StatStack and AET instead rely on reuse time distribution to approximate reuse distance distribution. Reuse time, which measures the number of references between a reference and its reuse, is a simpler metric to collect on the fly. The idea of spatial sampling in SHARDS is later extended to simulate any replacement policy of choice with a low overhead. This miniature simulation method [22] is directly adopted in this paper.

6 Conclusion

This paper presents a new replacement policy, DLRU, for Redis. DLRU is built upon the existing approx-K-LRU policy. Rather than fixing K across Redis execution, DLRU chooses an optimal K in every execution interval based on a cost model that estimates the miss penalty. We engineer a dynamic system using a low-overhead cache simulator. Experimental results demonstrate that it works well for both simplified and general conditions regarding object size, and can always match the best K performance or outperform a fixed-K system across a range of storage traces. To our best knowledge, DLRU is the first system to dynamically select a replacement policy along key-value cache execution to adapt to the access pattern changes. Our future work will take more replacement polices into consideration.

Acknowledgments

We would like to thank the anonymous reviewers for their constructive comments and suggestions. The research is supported in part by National Science Foundation CSR1618384, the National Key R&D Program of China under Grant No. 2018YFB1003505, and the National Science Foundation of

China (Nos. 61472008, 61672053 and U1611461), and Shenzhen Key Research Project No. JCYJ20170412150946024.

References

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

[2] Amazon. 2019. *Amazon Elastic Cache.* Retrieved Oct. 15, 2019 from https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/SelectEngine.html

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

[4] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: Miss-Ratio Curve Guided Partitioning in Key-Value Stores. In *Proceedings of the 2018 ACM SIGPLAN International Symposium on Memory Management* (Philadelphia, PA, USA) *(ISMM 2018)*. Association for Computing Machinery, New York, NY, USA, 84–95. https://doi.org/10.1145/3210563.3210571

[5] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2019. Faster Slab Reassignment in Memcached. In *Proceedings of the International Symposium on Memory Systems* (Washington, District of Columbia) *(MEMSYS '19)*. Association for Computing Machinery, New York, NY, USA, 353–362. https://doi.org/10.1145/3357526.3357562

[6] Jiqiang Chen, Liang Chen, Sheng Wang, Guoyun Zhu, Yuanyuan Sun, Huan Liu, and Feifei Li. 2020. HotRing: A Hotspot-Aware In-Memory Key-Value Store. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 239–252. https://www.usenix.org/conference/fast20/presentation/chen-jiqiang

[7] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 321–334.

[8] D. Eklov and E. Hagersten. 2010. StatStack: Efficient modeling of LRU caches. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. 55–65.

[9] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX Association, Santa Clara, CA, 57–69.

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

[11] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. 2018. Fast Miss Ratio Curve Modeling for Storage Cache. *ACM Trans. Storage* 14, 2, Article 12 (April 2018), 34 pages. https://doi.org/10.1145/3185751

[12] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, and Joel Emer. 2010. High Performance Cache Replacement Using Re-Reference Interval Prediction (RRIP). *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 60–71. https://doi.org/10.1145/1816038.1815971

[13] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. 1970. Evaluation Techniques for Storage Hierarchies. *IBM Syst. J.* 9, 2 (June 1970), 78–117. https://doi.org/10.1147/sj.92.0078

[14] memcached. 2018. *memcached.* Retrieved May 10, 2018 from https://memcached.org

[15] Mutilate. 2019. *Mutilate.* Retrieved Feb. 22, 2019 from https://github.com/leverich/mutilate

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

[17] Redis. 2019. *Redis 4.0.13.* Retrieved Feb. 22, 2019 from http://download.redis.io/releases/

[18] Redis. 2019. *Redis Data Types.* Retrieved Oct. 15, 2019 from https://redis.io/topics/data-types

[19] Redis. 2019. *Redis Deployment.* Retrieved Dec. 26, 2019 from https://redis.io/topics/whos-using-redis

[20] Redis. 2019. *Redis Replacement Policy.* Retrieved Oct. 15, 2019 from https://redis.io/topics/lru-cache

[21] TechStacks. 2019. *Redis Deployment Listed in TechStacks.* Retrieved Dec. 26, 2019 from ttps://techstacks.io/tech/redis

[22] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. 2017. Cache Modeling and Optimization using Miniature Simulations. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 487–498. https://www.usenix.org/conference/atc17/technical-sessions/presentation/waldspurger

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

[24] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas J. A. Harvey, and Andrew Warfield. 2014. Characterizing Storage Workloads with Counter Stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO, 335–349. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wires

[25] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. 2013. HOTL: A Higher Order Theory of Locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 343–356. https://doi.org/10.1145/2451116.2451153

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