pRedis: Penalty and Locality Aware Memory Allocation in Redis

Cheng Pan
Dept. of CS, Peking University, Beijing, China
Peng Cheng Laboratory, Shenzhen, China
ICNLAB, School of ECE, Peking University, Shenzhen, China
pancheng@pku.edu.cn

Yingwei Luo
Dept. of CS, Peking University, Beijing, China
Peng Cheng Laboratory, Shenzhen, China
ICNLAB, School of ECE, Peking University, Shenzhen, China
lyw@pku.edu.cn

Xiaolin Wang
Dept. of CS, Peking University, Beijing, China
Peng Cheng Laboratory, Shenzhen, China
ICNLAB, School of ECE, Peking University, Shenzhen, China
wxl@pku.edu.cn

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

ABSTRACT

Due to large data volume and low latency requirements of modern web services, the use of in-memory key-value (KV) cache often becomes an inevitable choice (e.g. Redis and Memcached). The in-memory cache holds hot data, reduces request latency, and alleviates the load on background databases. Inheriting from the traditional hardware cache design, many existing KV cache systems still use recency-based cache replacement algorithms, e.g., LRU or its approximations. However, the diversity of miss penalty distinguishes a KV cache from a hardware cache. Inadequate consideration of penalty can substantially compromise space utilization and request service time. KV accesses also demonstrate locality, which needs to be coordinated with miss penalty to guide cache management.

We propose pRedis, *Penalty and Locality Aware Memory Allocation* in Redis, which synthesizes data locality and miss penalty, in a quantitative manner, to guide memory allocation and replacement in Redis. At the same time, we also explore the diurnal behavior of a KV store and exploit long-term reuse. We replace the original passive eviction mechanism with an automatic dump/load mechanism, in order to smooth the transition between access peaks and valleys. Our evaluation shows that pRedis effectively reduces the average and tail access latency with minimal time and space overhead. For both real-world and synthetic workloads, our approach delivers an average of 14% ∼ 52% latency reduction over a state-of-the-art penalty aware cache management scheme, Hyperbolic Caching, and shows more quantitative predictability of performance.

CCS CONCEPTS

• **Information systems** → *Key-value stores*; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

penalty and locality aware, cache modeling, memory allocation

ACM Reference Format:
Cheng Pan, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2019. pRedis: Penalty and Locality Aware Memory Allocation in Redis. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362729

1 INTRODUCTION

Nowadays, more and more web applications need to exchange different types of data with the background servers. The front-end server might serve a wide variety of data requests by accessing multiple, off-site, replicated backend databases, such as multiple PostgreSQL [9], MySQL [7] clusters. Meanwhile, there is an increasing demand for short latency to access a large amount of data as used for gaming, E-commerce and others. An in-memory key-value (KV) cache system (like Redis [11], Memcached [3]) has become a performance-critical layer in today's client/server architecture, which aggressively caches objects from back-end stores. Unlike conventional caches, which are usually used to bridge the latency gap between layers of memory hierarchy, the KV cache can also be used to store data objects that represent expensive-to-compute values, such as results of popular database queries. Redis, which supports typed objects, becomes the first choice when various kinds of data need to be cached.

The performance of a caching system is highly determined by the *replacement algorithm*, that is to say, the strategy for prioritizing items for eviction when the cache is full. But in real-world scenarios, for the sake of simplicity and versatility, the common KV caches often resort to a recency-based replacement algorithm such as LRU in Memcached and an approximated LRU policy [1] in Redis. The hidden assumption is that the miss penalty is uniform. However, a KV cache can store objects from different sources [14]. For example, in a complex large web service, a cached object might come from a static page on a local disk, from a remote server, or from a local computation such as a statistic update in a time interval. That is to say, the cost or miss penalty of evicting different elements is different. The issue of miss penalty has drawn widespread attention. GreedyDual [41], GD-Wheel [30] and PAMA [33] all show that miss penalty discrepancy should affect cache replacement decisions and cache modeling. A recent penalty aware cache management scheme, Hyperbolic Caching (HC) [16], combines the miss penalty, access count and residency time of an item to deliver a better cache replacement scheme. HC shows its advantage over LRU on request service time. However, HC is short of a global view of access locality, either in short term or long term.

A Miss Ratio Curve (MRC), is an effective way to quantify cache locality. It reveals the relationship between cache size and cache miss ratio. We can use it to estimate *how much* data is being used by a particular workload and *what utility* can be gained by increasing its memory allocation [19]. This paper adopts a recent low-overhead, high-accuracy MRC model, EAET [34] to generate MRCs. EAET extends AET [27, 28] by modeling various operations, such as read, write, update and delete, on data items with different sizes. We integrate this model into a realistic Redis system and evaluate its effect. We will discuss EAET in Section 2.2. There are some earlier studies on optimizing memory allocation under the MRC guidance, e.g. LAMA [25, 26], Dynacache [22], Cliffhanger [23], mPart [19], but they are all based on slab reallocation in Memcached. Little has been done for Redis due to its complexity.

On a longer time scale, locality and data reuse can demonstrate a periodic access pattern. A study on Memcached workloads shows clear diurnal patterns [14]. The load variation during a diurnal cycle can be on a factor of two. If we can take advantage of this periodicity to guide data distribution, we can further improve Redis.

In this paper, we propose a real-time, penalty- and locality-aware memory allocation scheme, *pRedis*, which considers both data locality and miss penalty for memory allocation and object replacement in Redis. Experimental results on real-world and synthetic traces show 14 ~ 52% latency reduction over HC, on average. To summarize, we make the following contributions:

(1) We systematically design a new memory allocation scheme on top of Redis, which combines penalty and locality quantitatively, and delivers predictable cache performance.
(2) We propose the concept of the *penalty class* and implement a low-overhead penalty class mechanism on top of Redis using customized Bloom Filters [17].
(3) We implement our pRedis on top of Redis, evaluate the accuracy and overhead of EAET model, and demonstrate its performance benefits for several real-world and synthetic traces.

2 BACKGROUND

This section first illustrates LRU and HC through an example trace, and then compares them against pRedis where the impact of miss penalty and data locality is considered. We also introduce the EAET model adopted by pRedis.

2.1 Impacts of Miss Penalty

We define the *miss penalty* as the time interval between the miss of a GET request and the SET of the same key immediately following the GET [33]. We have analyzed several real-world Redis traces from a cloud computing provider. Figure 1 shows the distribution of the miss penalty for one trace (from an e-commerce application, 3 days of requests). It can be found that in this scenario, miss penalty can vary in a wide range. We will use a simple example to illustrate that ignorance of miss penalty in cache management policies may lead to many high-penalty misses and degrade cache performance.

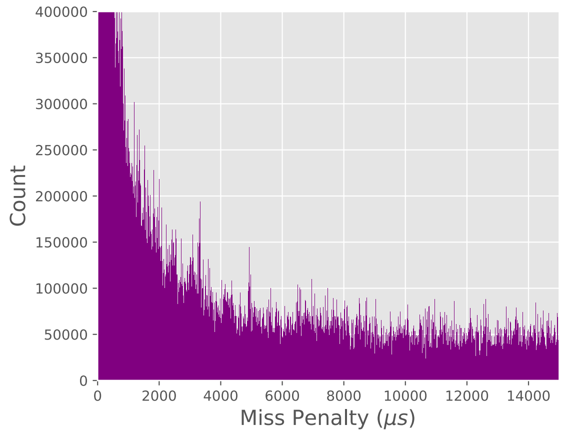
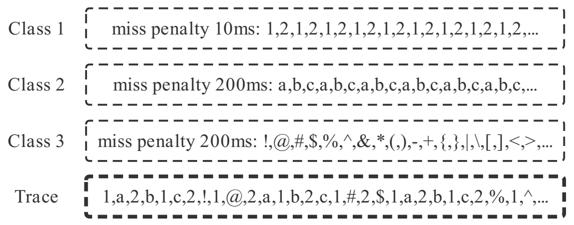


Figure 1: Miss Penalty Histogram in GET Requests



Class 1
miss penalty 10ms: 1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,1,2,...
Class 2
miss penalty 200ms: a,b,c,a,b,c,a,b,c,a,b,c,a,b,c,a,b,c,...
Class 3
miss penalty 200ms: !,@,#,$,%,^,&,*,(,),-,+,{,},|,\,[,],<,>,...
Trace
1,a,2,b,1,c,2,!,1,@,2,a,1,b,2,c,1,#,2,$,1,a,2,b,1,c,2,%,1,^,...

Figure 2: Three Different Miss Penalties

Assume that there are three classes of data accesses to Redis, and the miss penalties of the three classes are 10 ms, 200 ms, and 200 ms, respectively. As shown in Figure 2, the request sequence of class 1 is "1, 2, 1, 2, 1, 2, ...", the request sequence of class 2 is "*a*, *b*, *c*, *a*, *b*, *c*, ...", and the request sequence of class 3 is "!, @, #, $, %, ...", where the accesses of class 3 are all distinct. If the access rates of these three classes are 5 : 3 : 2, the combined access pattern can be "1, *a*, 2, *b*, 1, *c*, 2, !, 1, @, 2, *a*, 1, *b*, 2, *c*, 1, #, 2, $, 1, *a*, 2, *b*, 1, ...". We also assume that each item's hit time is 1 ms, and the total memory size is 5. Now we show how the LRU policy and Hyperbolic Caching (HC) handle this access sequence.

LRU Redis uses an approximated LRU (approx-LRU) strategy which randomly samples f items and evict the least recently accessed one. Studies have shown that the approx-LRU policy in Redis performs close to the LRU policy [1, 10]. The random sampling behavior is hard to illustrate. We instead use LRU to represent approx-LRU used in Redis in the whole paper. An LRU cache can be viewed as a stack. Data blocks are sorted from top to bottom by their most recent access time. Since the memory capacity is 5, the stack can hold at most 5 items.

As illustrated in Figure 3, every access to class 1 will be a hit (except for the two cold misses at the beginning), and other accesses

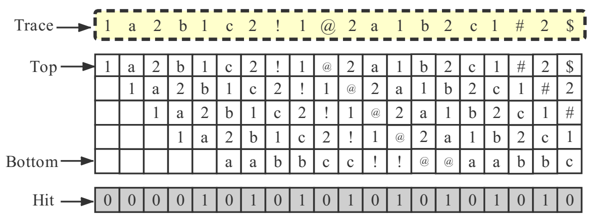


Figure 3: LRU Policy Processing Results

to class 2 and class 3 will all be misses. Given the miss penalty and hit time mentioned above, the average request latency can be calculated as $0.5 * 1 + 0.3 * (200 + 1) + 0.2 * (200 + 1) = 101$ ms.

Hyperbolic Caching In the Hyperbolic Caching (HC) approach, an item's priority considers its frequency and residency time since it entered the cache, and its fetching cost:

$$p_i = c_i * \frac{n_i}{t_i} \tag{1}$$

where n_i and t_i are the request count for i and the residency time of i, respectively, since it entered the cache, and c_i is the cost of fetching item i, i.e., the miss penalty. When the cache is full, HC selects an element with the lowest p_i to evict. All the state of item i will reset when it is evicted.

In Figure 4, we use x and y to represent the data items in Class 3 (every 10 requests contain 2 requests from Class 3.). We simply set c_i as the miss penalty of an item and use a logical clock to measure t_i. The blocks filled with red represent the evicted items. Because the miss penalty for class 2 and class 3 is much larger than class 1, the elements in class 1 are chosen to evict except for their first load. On the other hand, the newest class 3 elements stay in cache even there is no reuse. The average request latency with HC is $0.5 * (10 + 1) + 0.3 * 1 + 0.2 * (200 + 1) = 46$ ms.

Locality-Aware (pRedis) In HC, the access frequency and the time that an element resides in the cache are used to implicitly characterize the locality of data accesses: an element which has 1) more access count and 2) just entered the cache will have a higher priority. However, the high-penalty data with poor locality (i.e. class 3) will hold up the valuable cache space with the HC policy, even though they will not be used anymore. If we model and predict data locality using a Miss Ratio Curve (MRC), for each of the three classes, we can accurately predict the performance of the cache in different configurations and make eviction decision more wisely. We have:

$$mr_1(c_1) = \begin{cases} 1 & c_1 < 2 \\ 0 & c_1 \geq 2 \end{cases} \qquad mr_2(c_2) = \begin{cases} 1 & c_2 < 3 \\ 0 & c_2 \geq 3 \end{cases}$$

and $mr_3(c_3) = 1$, for any given cache size c_3, where c_1, c_2, c_3 represent the memory allocated to three classes, respectively.

The overall miss penalty can be expressed as $W = 0.5 * mr_1(c_1) * 10 + 0.3 * mr_2(c_2) * 200 + 0.2 * mr_3(c_3) * 200$, subject to $c_1 + c_2 + c_3 = 5$. When $c_1 = 2, c_2 = 3, c_3 = 0$, W will reach the minimum value of 40. Then the optimal result can be achieved: data items in class 1 and class 2 are always hit, and the average request latency is $0.5 * 1 + 0.3 * 1 + 0.2 * (200 + 1) = 41$ ms. If we can predict the MRC and access rate for each class, we can find the optimal memory allocation for different classes. This is a core motivation of pRedis which takes both data locality and miss penalty into account.

2.2 EAET

We apply EAET in pRedis to model data locality. The model is an extension of the AET model, and it can handle non-uniform data sizes and various operation types (read, write, update and delete). Both AET and EAET are based on the assumption: when the users encounters a `GET` miss, they will immediately re-fetch the data from backend database and `SET` it in Redis. From analysis of real-world Redis traces, we find that this assumption is true. This section first briefly describes the key insight of AET, and then the enhancement of AET.

2.2.1 AET. The AET model introduces a series of kinetic equations to model cache eviction process. The only input to the AET model is a Reuse Time Histogram (RTH), and the output is a Miss Ratio Curve (MRC). An item's (backward) reuse time is the time between the current access to this item and its last access. With RTH, we can calculate the probability that reference x has reuse time greater than t, which is then related to data movement in an LRU stack.

The AET model establishes a relationship between the reuse time distribution and the *average eviction time* (AET). AET zooms in on the eviction process of an evicting block from its last access to its eviction. During this process, the evicting block spends eviction time, $AET(c)$, on average, to travel c stack positions from top to bottom, and then gets evicted. $AET(c)$ can be calculated through the following equation:

$$\int_0^{AET(c)} v(t)\,\mathrm{d}t = c, \tag{2}$$

given $v(t)$, the traveling speed of an data block at time t. Hu et al. show that $v(t)$ is equal to the probability that a reference has reuse time greater than t:

$$v(t) = P(t) = \sum_{i=t+1}^{\infty} \frac{rt(i)}{N} \tag{3}$$

where N is the length of the sampled sequence and $rt(i)$ is the number of accesses with reuse time i. When the reuse time distribution is sampled, Equation 2 can find $AET(c)$. Finally, the miss ratio $mr(c)$ for an LRU cache with size c is indeed the probability that a reuse time is greater than the average eviction time:

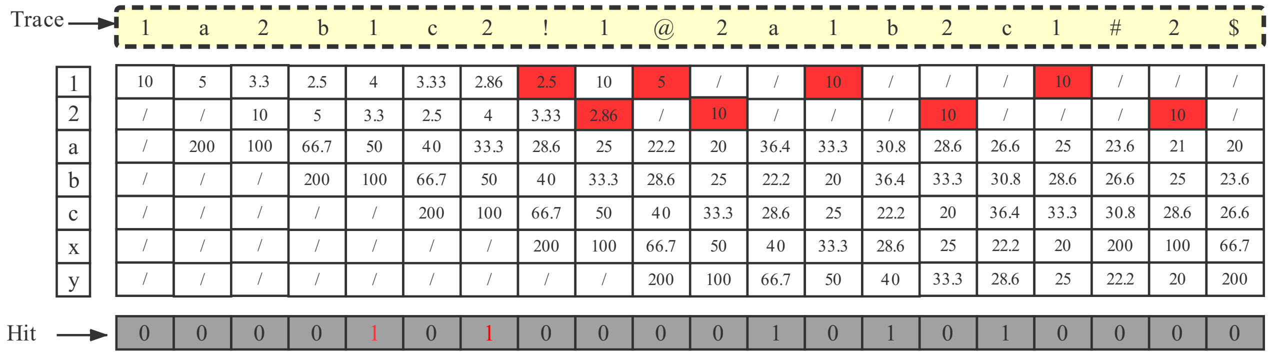
$$mr(c) = P(AET(c)) \tag{4}$$

2.2.2 Enhancement of AET. EAET enhances AET by dealing with two issues:

Issue 1: How to deal with non-fixed object size?
Issus 2: How to deal with various operation types?

The solutions to these two issues are quite intuitive.

(1) For **Issue 1**, EAET just decomposes the each operation to per-byte operation. For an operation like "`READ obj`", the decomposition will be "`READ obj[0], READ obj[1], . . . , READ obj[k-1]`", where `k` is the length of `obj`.



Trace	1	a	2	b	1	c	2	!	1	@	2	a	1	b	2	c	1	#	2	$
1	10	5	3.3	2.5	4	3.33	2.86	2.5	10	5	/	/	10	/	/	/	10	/	/	/
2	/	/	10	5	3.3	2.5	4	3.33	2.86	/	10	/	/	/	10	/	/	/	10	/
a	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	36.4	33.3	30.8	28.6	26.6	25	23.6	21	20
b	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	36.4	33.3	30.8	28.6	26.6	25	23.6
c	/	/	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	36.4	33.3	30.8	28.6	26.6
x	/	/	/	/	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	200	100	66.7
y	/	/	/	/	/	/	/	/	/	200	100	66.7	50	40	33.3	28.6	25	22.2	20	200
Hit	0	0	0	0	1	0	1	0	0	0	0	1	0	1	0	1	0	0	0	0

Figure 4: Hyperbolic Caching Policy Processing Results

(2) For **Issue 2**, EAET decomposes the RTH sampling to two kinds, (`access_rth[]` and `delete_rth[]`). It considers possible upward data movements due to deletions in an LRU stack. The movement of a data block can be bi-directional: 1) An downward move, caused by a set or get operation. (sample reuse time distribution in `access_rth[i]`); 2) An upward move, caused by a delete operation. (sample reuse time distribution in `delete_rth[i]`).

The overall speed of movement, $v(t)$ or $P(t)$), is just the superposition of the velocities in both directions:

$$v(t) = P(t) = \sum_{i=t+1}^{\infty} \frac{access\_rth[i]}{N} - \sum_{i=0}^{t-1} \frac{delete\_rth[i]}{N} \tag{5}$$

It is conceivable that if the deletion speed is greater than the access speed, the data block in the LRU stack tends to move upward as a whole. With the travel speed set (i.e. $P(t)$), Equation 2 can be applied to find the average eviction time, $AET(c)$, and Equation 4 derives the miss ratio curve.

3 PENALTY AND LOCALITY AWARE MEMORY ALLOCATION

This section details the design of pRedis. We first introduce penalty class and its implementation. We then present memory allocation driven by both locality and penalty.

3.1 Penalty Class Mechanism

In many applications, the miss penalty of items are related to one another. For example, some items may be created by database joins, while others are the result of simple indexed lookups. Rather than managing each KV individually, we can group them into different classes (i.e. *penalty class*) and within the same class, KVs share similar miss penalties. As mentioned in Sec 2.1, pRedis can reduce access latency thanks to accurate and appropriate memory partitioning. The first challenge is to keep track of the *penalty class* on top of Redis effectively.

Challenges The most intuitive idea is to store an additional value for each stored key to represent the penalty *class ID* (cid). Considering that the original data structure of Redis has been fully 8-byte aligned, adding even 1-byte variable will result in at least 8-byte overhead, which is quite unacceptable. Moreover, once an item is evicted from Redis, all the information it carries will be lost. That is to say, we don't know what the class IDs of the re-fetched items are. If we store all the keys we have visited to 'remember' all class IDs, the space overhead is too high (even beyond the space needed to store the original data).

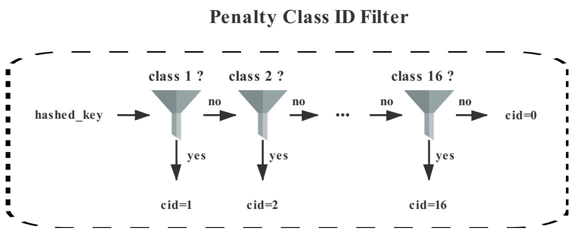


Figure 5: Chain-structured Bloom Filter for Penalty Class

Our simple solution in pRedis is shown in Figure 5, which is a Bloom Filter chain for discrimination of penalty class IDs. Each filter needs to answer "Does this key belong to class i?", for i from 1 to N, where N is total number of penalty classes. If anyone's answer is 'yes', then we get the cid.

There are four factors in each Bloom Filter, $BF(m, k, p, n)$, where m denotes the length of the bit array, k denotes the number of hash functions, p denotes the probability of classification error, and n denotes the total number of elements expected to be inserted. In the case where p and n are known, the most suitable m and k can be calculated as follows:

$$m = -n * \frac{\ln p}{(\ln 2)^2} \tag{6}$$

$$k = \frac{\ln p}{\ln 2} \tag{7}$$

In actual use, we set p to 0.01. When n is 1 million, a Bloom Filter takes up about 1 MB. At the same time, it takes about 1 μs to find the cid of a key, which is relatively low compared to object access time.

Considering that Redis' access latency to a small KV is around 20 μs, we further improve our Class ID Filter to reduce its time consumption. In Figure 6, we build a Bloom Filter tree for penalty

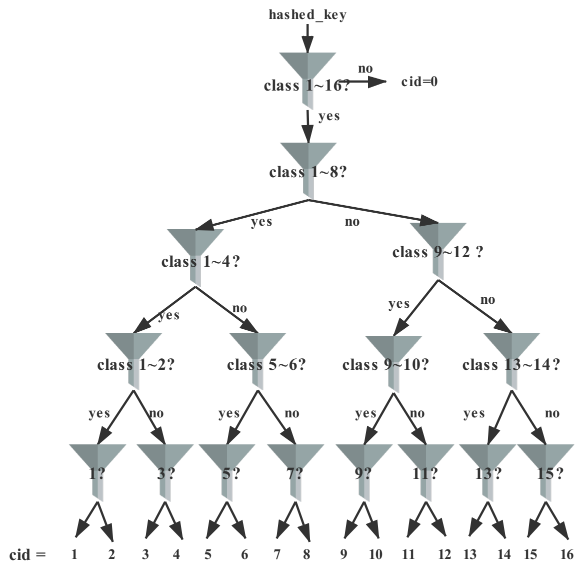


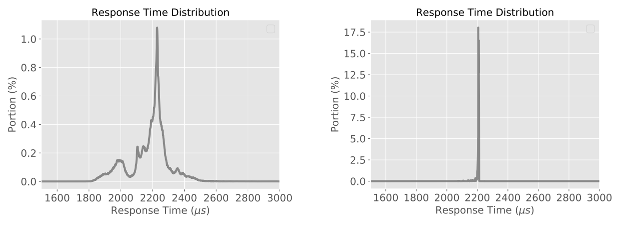
Figure 6: Tree-structured Bloom Filter for Penalty Class

class discrimination. Every key needs to travel only 5 filters to get the cid, comparing to 8 filters on average for the filter chain in Figure 5.

3.2 Determining a Penalty Class

In this section, we consider how to determine a key's penalty class. The items in the same miss penalty class imply they have similar miss penalties, and we provide two kinds of method to set an item's class: auto-detecting and user-hinted.

Auto-detecting Penalty Class A simple way to partition different key-value pairs is to set the range of each penalty class in advance (similar to size class in Memcached [2]), and then each KV will be automatically assigned to the class it belongs to based on the measured miss penalty. We observe exponential distributions of miss penalty in real-world Redis traces (see Figure 1). We thus increase the range of each class exponentially, similar to the choice of size class range in Memcached.



(a) single latency measurements (b) aggregate latency measurements

Figure 7: Two Kinds of Miss Penalty Measurement

User-hinted Penalty Class Although the way to automatically detect the penalty class is very simple, there are some drawbacks. The measured latency to access the same item is stochastic. Some requests might experience longer delays than others and thus are put into a different class, even though the higher cost has nothing to do with the item itself [16]. We use MySQL as the backend database of pRedis, and measure the latency when the same key is missed. After repeating this experiment 10,000 times, we plot the distribution of the measured miss penalty shown in Figure 7a. One can find that the latency typically ranges from 1800 μs to 2600 μs, and shows a significant variation. The way to solve this problem is to adopt a user hint. Users who use pRedis as a cache often know where to fetch data after the cache miss. At the same time, users may know whether the miss penalties of some objects are similar as they are from the same locations or same data structures.

Our system provides an interface for a user to specify the class of an item. We then set/reset the miss penalty for a class on the fly periodically. We aggregate the latency of all items of a penalty class in a time period and adopt the average as the penalty for the class. This results in a more unified penalty as shown in Figure 7b. Another advantage of this periodical aggregation comes from rapid adaption to massive changes of miss penalty, e.g., a replica of a database fails. The auto-detecting method would only update the individual costs after the items have been evicted, one by one. But the user-hinted aggregating penalty method will quickly reset the penalty. What we need to emphasize here is user-hint is optional. If the user can provide accurate hints, user-hints will be a better option. Nevertheless, auto-detecting mode can also deliver good performance based on our evaluation.

3.3 MRC-based Memory Allocation

In this section, we describe in detail how to allocate memory among different penalty classes, using locality and penalty to achieve optimal service latency.

In order to deal with access pattern changes, we divide a workload into fixed time-windows (or phases) according to the number of requests. The length of a time-window can be configured according to the workload (default is set to 1 million accesses). Three steps are followed in each time window:

Step 1: RTH Sampling We use set sampling technique [27] to sample the reuse time histograms (RTH) of *access operations* and *delete operations* for each penalty class.

Step 2: MRC Construction At the end of the time window, we feed the collected RTHs (`access_rth[], delete_rth[]`) of each penalty class into the EAET model, which can generate an MRC for each penalty class in linear time.

Step 3: Memory Reallocation If we allocate penalty class i with M_i memory, then this class's overall miss penalty (or latency) MP_i can be estimated as:

$$MP_i = mr_i(M_i) * p_i * N_i, \tag{8}$$

where N_i is access frequency of class i, p_i is the average individual miss penalty, and $mr_i(M_i)$ is the miss ratio at memory size of M_i, which can be derived from the MRC. Our final goal is to minimize the total miss penalty for all penalty classes:

$$min \sum_{i=1}^{n} MP_i = min \sum_{i=1}^{n} mr_i(M_i) * p_i * N_i \qquad (9)$$

$$s.t. \sum_{i=1}^{n} M_i = M, \qquad (10)$$

where n is the number of penalty classes and M is total memory we can allocate. The optimal solution can be reached through dynamic programming with cost function:

$$f[i][j] \leftarrow \min_{k}\{f[i-1][k] + MR_i[j-k] * p_i * N_i\} \qquad (11)$$

where $f[i][j]$ represents the minimal miss penalty when the first i classes are allocated j bytes.

The time complexity of the dynamic programming algorithm is $O(nM^2)$. M will be a large value (in bytes) in actual applications, but we usually do not allocate memory in bytes, but at a larger granularity G (such as 1 MB, like default slabs size in Memcached). So the final time complexity is $O(n(M/G)^2)$, which becomes more practical in real systems.

Fairness The cost function (Equation 11) doesn't consider the fairness between different penalty classes, which could lead to higher miss rates for some penalty classes. In our design, fairness is not a concern at the level of memory allocation, because we sacrifice fairness for efficiency. If fairness is a concern, we can amend the cost function to discard an unfair scheme and optimize the performance under the fairness constraint. A recent solution is the baseline optimization by Brock et al. [18] and Ye et al. [40].

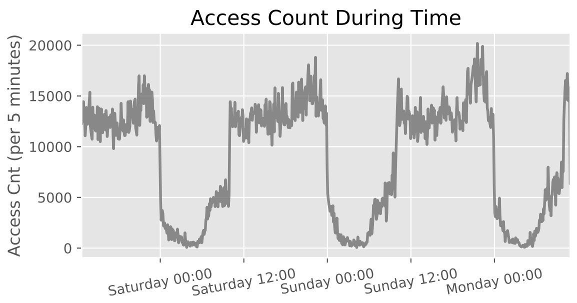
4 LONG-REUSE LOCALITY HANDLING

We consider two aspects of data locality:

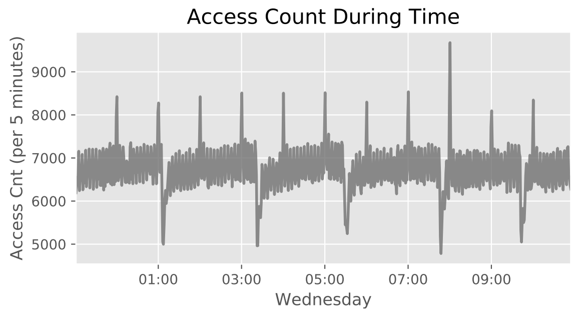
- **Short-term locality (minutes-level).** We use a time window (every million accesses, several minutes) to trigger reallocation according to online MRCs just as we have discussed in Sec. 3.3. The window is quite short compared to days/months of KV store execution.
- **Long-term locality (hours-level).** That is to explore the diurnal behavior of KV stores. "Auto dump/load" is designed to handle this situation, we replace the passive eviction mechanism with an automatic dump/load mechanism, to smooth the transition between access peaks and valleys. We will discuss the details below.

On a longer time scale, locality and reuse can present a periodic access pattern: 1) the load variation during a diurnal cycle can be very significant; 2) during consecutive demand peaks, there are always some data being reused. We have analyzed this long-term reuse through a collection of 16 Redis traces from a cloud computing provider. These traces send requests from different source IPs to the Redis server over several days. We analyze the distribution of the number of requests sent from different IPs over time and find that there are mainly two types of workload:

Periodic Pattern: The number of requests changes periodically over time, and the long-term reuse is accompanied by the emergence of request peaks.
Non-periodic Pattern: The number of requests remains relatively stable over time, or there are no long-term reuses.



(a) Periodic Pattern



(b) Non-periodic Pattern

Figure 8: Two Types of Requesting Pattern

We pick 2 representative examples out of these 16 traces, shown in the Figure 8. Figure 8a demonstrates a strong diurnal behavior. We also observe that, during every peak, the requests always access a fixed set of data. In contrast, in Figure 8b, we can hardly see the changes in request pattern over time, and there are few long-term reuses.

Obviously, when these two types of workloads share Redis, with the LRU (approx-LRU) strategy, the memory usage of the two types of data will change during the access peaks and valleys. However, the passive evictions during the valley periods and the passive loadings (because of GET misses) during the peak periods will cause considerable latency.

Auto-Dump/Load in Periodic Pattern We solve this problem using dump/load during access peaks and valleys. We will proactively dump some of the memory to a local SSD (or hard drives) when a valley arrives, and before arrival of a peak, proactively load the previously dumped content. We can then reduce the time required for passive eviction and the time it takes to get data from the back-end databases as needed for passive loadings.

To be more specific, when the access pattern of a penalty class steps into a valley, we use the LRU strategy to dump the KVs of the class to the external storage in a sequential manner. At the same time, for the sake of consistency, we need to support modification and deletion of the dumped KVs, so we must also retain the keys of those dumped KVs in memory. Any modification or deletion will be tagged. We will merge the results when the dumped files are loaded.

When it is detected that the access rate of a penalty class that has been dumped to external storage suddenly starts to increase, the auto-loading mode is entered (kind of large-scale prefetching). The loading process is a process that can be interrupted at any time, and the efficiency of Redis services won't be reduced.

We use several metrics to decide if a periodic pattern is present for a penalty class in the request sequence:

Peak and Valley Existence: the maximum and the minimum request rates are significantly different from the average request rate (e.g. max - min > 0.5 * avg).
Peak and Valley Duration: request rates at peaks and valleys must last for a continuous period of time (e.g. at last 10 minutes).
Long Reuse Time Length: the long reuse time must be at the same magnitude of the peak-to-peak interval.

Only when all three conditions are met, we decide that the class has a periodic pattern. We count the accesses of each penalty class for every 60 seconds. If the number of accesses is very close to the minimum access count in the past 10 minutes, the class is considered to have reached a valley state, and the class should be dumped to external storage. After that, if a dumped class encounters a large number of consecutive misses (we use 100 in our experiments), the class should be loaded back.

Auto-dump/load mode is our solution to handle the diurnal nature of workloads, which exploits the long-term locality. By dividing the locality handling into two parts (short-term and long-term), our optimization will be more effective.

5 IMPLEMENTATION OF PREDIS

On top of Redis-4.0.13, we have implemented our pRedis system. Redis supports the module system from version 4.0, so we can add our pRedis without changing the Redis kernel service. In Figure 9, we show an overview of pRedis structure. The total implementation requires 1780 lines of C code.

We use an example to illustrate the workflow in Fig. 9. Assume that there is a Redis request, `SET key 123`. Redis will look up the command handling table to find which function can be used to handle this request. Then if this command is memory-related, Redis will conduct a memory eviction check: If the max-memory limit is reached, the KV eviction will be invoked. Finally, Redis runs `cmd->proc()` to process the request and give the respone to the client. Our pRedis mainly cares about the three steps in the event loop mentioned above: command dispatcher (to monitor the operation type), memory eviction (to inject our memory allocation scheme) and cmd->proc (to monitor the execution result of the request and corresponding time).

We will describe the main pRedis modules in details below, including some challenges we faced and how we solved them.

pRedis Driver. The driver is an interface with two main functions: forwarding Redis commands to the pRedis module and managing the eviction strategy within Redis. Whenever Redis receives a command, the driver is responsible for forwarding the type of the command (read, write) and the corresponding key to the pRedis module, and then the pRedis module determines whether to sample the RTH and update the memory occupied by each penalty class. When the class memory allocation module performs a memory reallocation at the end of each phase, the driver receives a new allocation scheme and uses the scheme to guide the data eviction. When items need to be evicted, Redis samples `conf.maxmemory-samples` KV items from the key space, and chooses the one with the longest idle time to evict. We extend this policy by checking whether the chosen key belongs to a class whose used memory is larger than the suggested. If so, the eviction happens, otherwise we check the key with the second longest idle time, ans so on. If the first round sampling doesn't hit any item, we will retry the sampling process until an item is chosen by our method. The default `conf.maxmemory-samples` equals to 5, which has already delivered quite good results. We observe that 10 or more will approximate very closely to real LRU.

KV Size Cache. The EAET model needs to use item sizes. The Redis supports key-value size estimating since version 4.0, but it may cause some latency if we call this estimating function too frequently. So we set a fixed-size direct map cache to store kvsize. Each entry in the cache stores the hash value of the key and the corresponding KV size. Each entry occupies 4 bytes, including hash_val: 16 bit and KV_size: 16 bit. If the KV doesn't hit the cache, then we will call the kernel's size estimating function and store it to the direct mapping cache. We use the most significant bit of KV_size to indicate whether the size is in a unit of bytes or 32 KB (2^{15} = 32 KB). If the KV size is larger than 32KB, then it will be stored in multiples of 32KB. The largest size can be represented by this method is 1 GB ($2^{15} * 2^{15}$ =1 GB), which is sufficient for most scenarios.

Miss Penalty Monitor. According to our definition of miss penalty, a very intuitive idea to track the miss penalty is: when a key gets a miss, record the timestamp immediately. When the next time the key is set, the miss penalty is obtained by subtracting the two timestamps. But there is a very subtle problem here. The key that gets a miss has no entry in Redis key-space dictionary, so recording its information requires an extra entry. From another perspective, the element of each miss will be ultimately added to Redis, so we allocate memory to the element in advance and record the timestamp.

EAET Model We use the EAET algorithm described in Section 2.2 to generate the MRCs. With the *class_id* of each data item determined, we can sample reuse time for each class. Since we have obtained the memory size of each key-value pair in the size tracking module, `access_rth[]` and `delete_rth[]` can be calculated in byte level for each class. We use Equation 5 to generate $P(t)$ and use Equation 2 to get $AET(c)$. Finally, $mr(c)$ can be derived by

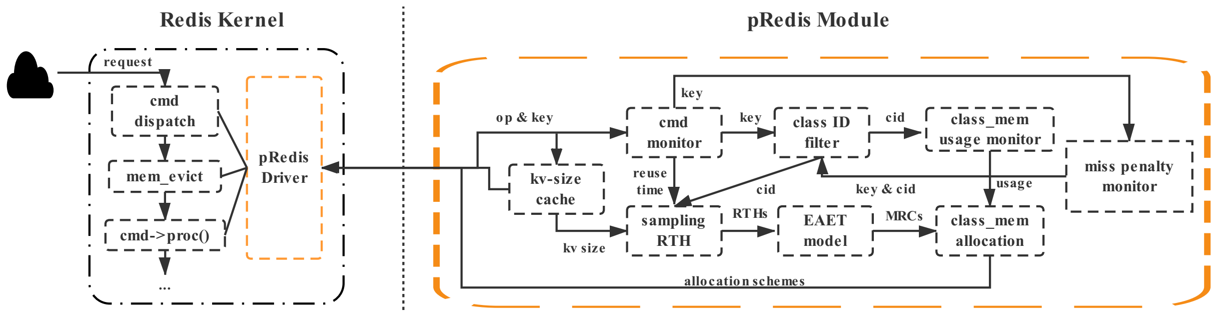


Figure 9: pRedis System Structure Overview

Equation 4.

Class Memory Allocation. In Section 3.2, we have discussed how to classify KVs according to their miss penalty. Now we need to consider how to support a low-cost memory partitioning in Redis. Our method is to modify the Redis eviction function. We maintain two arrays to maintain the amount of memory used in each class and the suggested memory allocation by our locality- and penalty-aware algorithm. Our experiments show that in the case of a small number of classes (e.g. $n \leq 16$), the time and space consumption of this modification on the eviction function is negligible as we will show in the evaluation section next.

6 EVALUATION

6.1 Experimental Setup

6.1.1 System Configuration. We evaluate pRedis and other strategies using six cluster nodes, each of which is configured with an Intel(R) Xeon(R) E5-2670 v3 2.30GHz processor with 30MB shared LLC and 200 GB of memory. The operating system is Ubuntu 16.04 with Linux-4.15.0. In order to evaluate the miss penalty, we set up two MySQL databases as the backend stores. One is deployed on the local host and the other is on a remote LAN node. pRedis supports clustering mode. We set up six pRedis instances (three masters, three slaves) for all experiments. The logical structure is shown in Figure 10. The response time is the wall-clock time spent by each client request, including the cost of the database access.

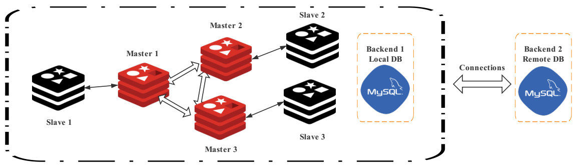


Figure 10: The Experimental Cluster Structure

6.1.2 Workloads. There are four suites of workload used for different aspects of the evaluation:

MSR Workloads MSR Cambridge traces [5] represent one week of block I/O traces from the Microsoft Research Cambridge Enterprise servers. We use the MSR traces in pRedis to evaluate the MRC accuracy. In the evaluation, the raw MSR traces are transformed to Redis operations with SET and GET.

YCSB Workloads Yahoo Cloud Serving Benchmark [12] (YCSB) is a framework and common set of workloads for evaluating the performance of different "key-value" and "cloud" serving stores. We use YCSB to evaluate the request response time (latency) by pRedis and compare it with Redis and Redis-HC.

A Collection of Real World Redis Workloads The Redis traces are from a cloud computing service provider, and they were obtained from a set of Redis servers used for E-commerce, cluster performance monitoring, and other services. The servers are tracked in a few days of different time periods and the traces record the original Redis operations such as `lpush, zset, zadd, lrange`, etc.

Memtier Benchmark Memtier_benchmark [4] is a high throughput benchmarking tool for Redis and Memcached. Memtier_benchmark is capable of launching multiple worker threads (using the `-t` option), with each thread driving a configurable number of clients. We will use this benchmark to evaluate the throughput of pRedis.

6.2 MRC Accuracy

In order to achieve high performance, pRedis relies on accurate MRCs. In this part, we will compare the pRedis MRC, obtained by EAET using 1% set sampling, with the actual MRC, obtained by measuring the full-trace reuse distances.

We use the MSR traces consistent with PACE [34] to evaluate the MRC accuracy of the EAET algorithm working in pRedis. The reuse distances are collected by an LRU simulator which supports various object sizes in cache replacement.

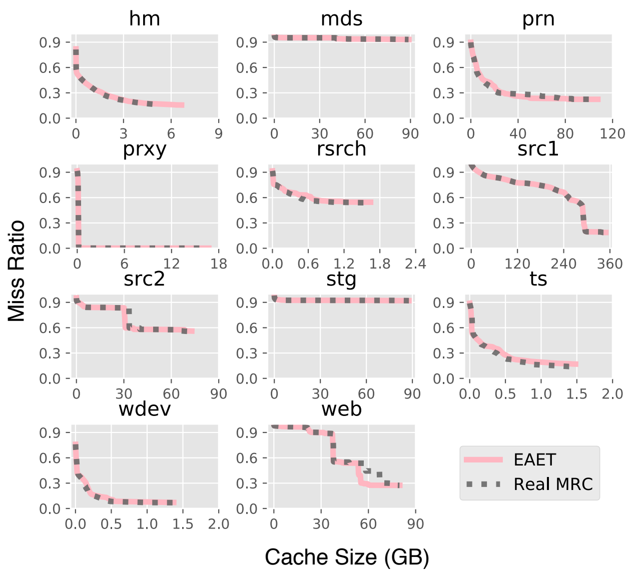
Figure 11 shows that EAET can achieve a high accuracy on MRC construction. The average absolute error of EAET is 1.2%, which is accurate enough for further cache performance optimization.

6.3 Latency

6.3.1 Average Latency. Our memory allocation strategy is focused on reducing request response time or latency. We use the YCSB workloads to evaluate the latency of default Redis, Redis-HC and pRedis. The workloads are detailed in Table 1. The record count of each workload is 1 million, the operation count is 500 million,

Table 1: YCSB Workloads Details

Workload	Feature	Examples	Parameter
A	Update heavy	Session store recording recent actions	read=0.5; update=0.5
B	Read heavy	Photo tagging, but most operations are to read tags	read=0.95; update=0.05
C	Read only	User profile cache, where profiles are constructed elsewhere (e.g. Hadoop)	read=1.0
D	Read latest	User status updates; people want to read the latest	read=0.95; insert=0.05
E	Short ranges	Threaded conversations, where each scan is for the posts in a given thread	scan=0.95; insert=0.05
F	Read-modify-write	User database, where user records are read and modified by the user	read=0.5; read_modify_write=0.5



	hm	mds	prn	prxy	rsrch	src1
abs err	0.58%	0.06%	1.9%	0.06%	1.6%	0.05%
	src2	stg	ts	wdev	web	**avg**
abs err	1.2%	0.07%	3.0%	1.1%	3.6%	**1.2%**

Figure 11: MRC Accuracy on MSR Traces

and both the request key distribution and the value's field length distribution are subject to Zipfian [13] distribution.

We use the MurmurHash3 [6] function to randomly distribute the data to two backend MySQL servers, one local and one remote. The access latency to the local and remote nodes are about 120 μs and 1000 μs, respectively.

We set the Redis max-memory limit to 50% of the working set size, and run Redis, Redis-HC and pRedis, respectively. Our pRedis has two variants: pRedis(auto), which uses auto-detecting penalty class; and pRedis(hint), which uses user-hinted penalty class. Here we need to clarify how we set a penalty class. For pRedis(auto), we set a series of ranges, [1μs, 10μs), [10μs, 30μs), [30μs, 70μs), ..., [327670μs, 655350μs), 16 penalty classes total. If the miss penalty (latency) belongs to any range, then the penalty class is decided.

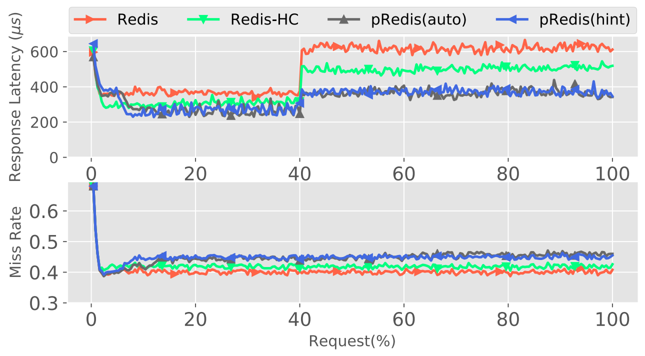


Figure 12: Latency and Miss Rate over Time

For pRedis(hint), we add an interface in pRedis to set the penalty class id for each item manually. Additionally, Redis-HC's cost factor (c_i in Eq. 1) is set to the actual miss penalty (latency) monitored by our miss penalty monitor module.

We choose workload A in YCSB to illustrate the latency reduction by pRedis. Additionally, in order to compare two different variants of pRedis, we run a stress test (mysqlslap [8]) in the remote MySQL server after the workload reaches 40% of the trace, causing the original latency to rise from 1000 μs to approximately 2000 μs.

We can see the access latency and miss rate over time in Figure 12. There are a few important points that we should pay attention to.

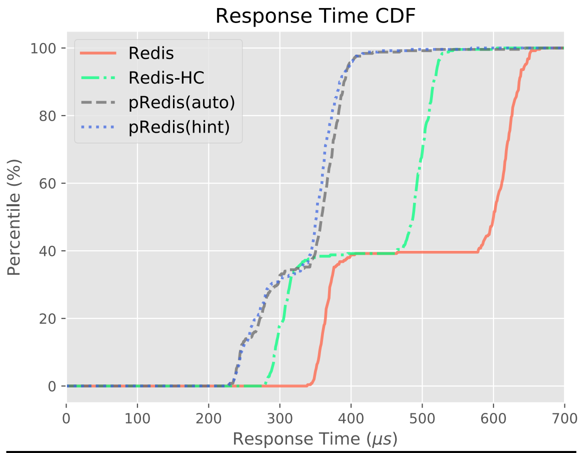
1) Request 10%. Prior to this point, pRedis is profiling for RTH and miss penalty without any interference on memory allocation, so both variants of pRedis have the same access latency and miss rate as Redis. At the same time, Redis-HC reaches lower access latency faster because its prioritized eviction scheme can take an immediate effect.

2) Request 10%-15%. At the point of 10%, we build an MRC for each penalty class using the collected RTH and calculate the optimal memory allocation scheme using dynamic programming. The memory allocation scheme takes effect gradually as we use the lazy eviction mechanism for memory partitioning. At 15%, the stable state is reached and the access latency becomes lower than Redis-HC. Note that the miss rate of pRedis is higher than those of Redis and Redis-HC. However, a higher miss rate does not imply a higher latency. The optimization target is performance, not miss rate.

3) Request 40%. Because the remote MySQL performs the stress test, causing rising delays, both Redis and Redis-HC observe a large increase in latency. pRedis gains benefits from accurate data locality and captured miss penalties, and thus only experiences a moderate latency increase.

We show the Cumulative Distribution Function (CDF) of response latency for the four schemes in Figure 13. pRedis can significantly reduce access latency compared to Redis and Redis-HC. If we compare the average response latency, pRedis(auto) is 34.8% and 20.5% lower than Redis and Redis-HC, respectively. pRedis(hint) cuts another 1.6%. The reason why pRedis(hint) is better than pRedis(auto) is that pRedis(auto) divides data objects into fixed penalty intervals on the fly. The classification is not as accurate as the user hints.

We summarize the average response latency of the six YCSB workloads in Figure 14. The average latency improvement by pRedis(auto) is 25.2% ~ 49.7% and 12.1% ~ 51.9% when compared to Redis and Redis-HC, respectively. Similarly, pRedis(hint) delivers a 26.8% ~ 50.1% and 14.0% ~ 52.3% improvement.



	Redis	Redis-HC	pRedis(auto)	pRedis(hint)
Avg (μs)	519.0	425.4	338.1	332.7

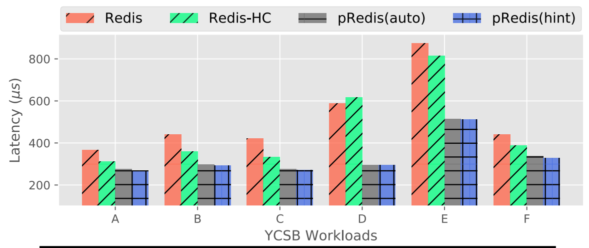
Figure 13: Access Latency CDF

6.3.2 Tail Latency. We have shown that pRedis reduces the average response time. A question is whether the pRedis overhead affects some requests disproportionally.

In Figure 15, we show the tail latency of pRedis, compared with Redis and Redis-HC. The workload is YCSB workload A, and memory limitation is 50% of the working set size. For about 99.99% of the response times in pRedis are the same as or lower than Redis and Redis-HC. From 99.999% to 99.9999%, three methods have their pros and cons. And for the next 0.00009% requests, pRedis performs better than others, which may be due to the fact that items of high-penalty are allocated more space.

6.4 Auto-Dump/Load in Periodic Pattern

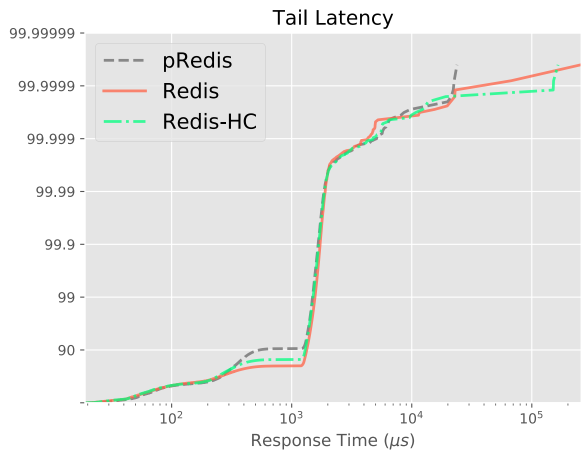
For the sake of simplicity, we eliminate a variety of interference factors, and design a simple experiment to test the effect of periodic diurnal access pattern and thrashing pattern on pRedis. We use two traces from the collection of Redis traces, one trace has periodic pattern (the e-commerce trace), and the other has non-periodic pattern (a system monitoring service trace). The data objects are also distributed to both the local and remote MySQL databases.



	pRedis(auto)					
	A	B	C	D	E	F
Redis	25.7%	33.6%	34.9%	49.7%	40.1%	25.2%
Redis-HC	12.8%	18.3%	17.6%	51.9%	35.9%	12.1%

	pRedis(hint)					
	A	B	C	D	E	F
Redis	26.8%	33.6%	35.6%	50.1%	41.3%	26.8%
Redis-HC	14.0%	18.3%	18.5%	52.3%	37.1%	15.0%

Figure 14: Average Response Latency Reduction in YCSB



	min	90th	95th	99th	99.9th	99.99th
Redis	19	1335	1415	1561	1712	1789
Redis-HC	19	1295	1369	1506	1664	1766
pRedis	19	540	1349	1490	1647	1747

Figure 15: Tail Latency Details

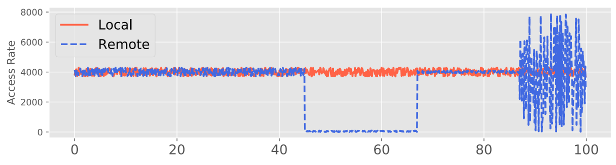
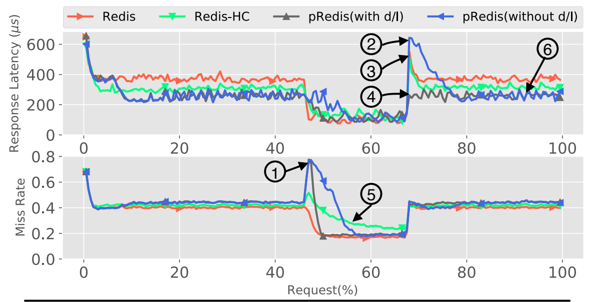


Figure 16: Periodic Access Pattern

We set the access rate at the same frequency, and design the access pattern as shown in Figure 16, the accesses to the data stored in the remote database pause at Request 45%, while the accesses to the data in the local database keep going. The remote accesses resume at Request 65%. The last 15% of the remote accesses are going to thrash (but the average access rate is still 4000). We show the access latency and miss rate over time in Figure 17.



	Redis	Redis-HC	pRedis (with d/l)	pRedis (without d/l)
Avg (μs)	317.3	277.2	240.3	262.6

Figure 17: Latency and Miss Rate over Time with Auto-Load/Dump

In Figure 17, we use *pRedis (with d/l)* to indicate that the automatic dump/load operations are enabled, and *pRedis (without d/l)* no automatic load/dump operations. At about Request 45%, the access to remote MySQL come into "valley" (paused), and after that (at about 65%), the access resumes, which means going back to "peak". We now discuss five interesting observations labeled in Figure 17.

Point ① The miss rate of the two pRedis variants suddenly rises because the accesses to the remote node suddenly stop, and in the allocation scheme, a large amount of space is allocated to this class, resulting in less space for the other class (which causes unfairness). However, the miss rate of pRedis(with d/l) drops quickly after, because the access valley is identified by the period detecting module, and the memory is dumped actively, but pRedis(without d/l) needs to wait until the end of this time window to perform memory re-partitioning.

Point ② pRedis(without d/l) does not detect peaks, so memory re-allocation will not occur until the end of the current time window is reached.

Point ③ The original Redis evicts all the data from remote MySQL because of the LRU policy, so it needs to reload them back completely.

Point ④ pRedis(with d/l) detects the peak and quickly load the original dumped data from the disk, so the latency transition is smooth.

Point ⑤ Redis-HC responds slowly to the termination of remote accesses. Because the miss penalties of this class of data are relatively high, the priority of this class takes time to degrade.

Point ⑥ The thrashing access pattern does not affect auto dump/load handling. The dump/load will not be triggered in this situation. The access latency is smoothed out because the running average is plotted.

In general, the use of auto-dump/load can smooth the access latency caused by periodic pattern switching, and active dump/load actions have significant advantages over passive eviction and re-fetching. pRedis(with d/l) is 24.3% and 13.3% lower than Redis and Redis-HC, respectively, in average response latency. Further, pRedis(with d/l) outperforms pRedis(without d/l) by 8.4%.

6.5 Overhead

6.5.1 Time Overhead. Among the new modules of pRedis, most time are spent to sample RTH, construct an MRC for each penalty class, and apply dynamic programming to obtain an optimal memory allocation scheme. This section focuses on testing the time overhead of these three functions.

Table 2: RTH Sampling (per item), MRC Construction(per penalty class) and Re-allocation DP Time Consumption

Phase No.	RTH (μs)	MRC (μs)	DP (μs)
1	0.87	275	118
2	1.19	288	97
3	1.52	282	109
4	1.87	274	109
5	1.60	273	98
6	1.71	272	95
7	2.01	301	120
8	1.89	290	112
avg	**1.58**	**280**	**107**

We use one of the real-world Redis trace to do the evaluation. In Table 2, we list the time consumption of the RTH sampling, the MRC construction and the re-allocation DP algorithm for the first 8 phases. The RTH sampling time includes the KV size tracking time and hash table maintaining time. Since the average response latency is about 150 μs per access, and the sampling rate is 1%, so sampling time takes about 0.01% of access time, which is acceptable. The MRC construction and re-allocation DP occur at the end of each phase, and the length of each phase is measured in minutes (which is configurable). A few hundred μs are negligible on a minute scale. In general, the time overhead of each module in pRedis is negligible.

6.5.2 Space Overhead. Table 3 lists the space overhead of pRedis when the working set is 10 GB (using YCSB Workload A). The default record size in YCSB is 1KB (10 fields, 100 bytes each, plus the key) and the number of records is about 10 million. pRedis uses 1% set sampling so the number of sampled records will be 100K. Each entry in the sampling hash table occupies 32 bytes (including the key pointer, last access time, etc).

The total space overhead is 25.08 MB, 0.24% of the total working set size. It should be mentioned that the space for RTH Arrays, MRC Arrays, Class IDs Filter and KV Size Cache are fixed, regardless of the working set size. In practical applications, the space overhead is acceptable compared to the benefits of these additional data structures.

Table 3: Space Overhead of pRedis

	Space Details
Sampling Table	100 K * 32 B = 3.2 MB
RTH Arrays	120 KB * 16 = 1.88 MB
MRC Arrays	1 K * 4 B = 4 KB
Penalty Table	16 * 4 B = 64 B
Class IDs Filter	1 MB * 16 = 16 MB
KV Size Cache	1 M * 4 B = 4 MB
Total	25.08 MB

6.6 Throughput

As the average response latency is reduced, the overall throughput of the system increases. We evaluate the throughput using the YCSB workloads, and the results are shown in Table 4. Here, pRedis represents pRedis(hint), which improves the throughput by 26.1 ∼ 100.0% and 11.5 ∼ 112.5% over Redis and Redis-HC, respectively.

Table 4: Throughput (kreq/s) in YCSB Workloads

	A	B	C	D	E	F
Redis	2.7	2.3	2.4	1.7	1.2	2.3
Redis-HC	3.2	2.8	3.0	1.6	1.2	2.6
pRedis	3.7	3.4	3.7	3.4	2.0	2.9

In the last experiment, we conduct a stress test using Memtier Benchmark. The memory-limit is set to ∞, so all of the GET queries will be hits. We setup 2 to 10 threads to send requests, each thread will drive 50 clients, each client send 1000000 requests total. The ratio of SET and GET is 1:10, and default data size is 32 bytes.

Table 5: pRedis vs. Redis on Throughput

Threads	2	4	6	8	10
Ratio (%)	97.8	97.5	98.6	98.8	99.5

We use different number of threads to send requests, repeat each test for 10 times, and show the average throughput ratios in Table 5. The purpose is to test the degradation when there are no misses in Redis. Although the throughput of pRedis is lower than the default Redis in the stress test, the average degradation is only 1.5%. In actual use, as long as there are significant misses, pRedis can divide data objects into penalty classes, perform optimized memory allocation, and show its performance advantage.

7 RELATED WORK

Peanlty Aware Caching Penalty- or cost-aware caching is a recent research hot topic. Some cost-aware strategies have incorporated properties such as freshness (e.g., [35]), which is similar to expiration time but not as strict. GreedyDual [41] attempts to incorporate cost into LRU, but it requires a re-design of the eviction strategy. Cao and Irani [20] implement GreedyDual using priority queues for size-aware caching in web proxies. GDWheel [30] implements GreedyDual in Memcached using a more efficient wheel data structure. PAMA [33], also implemented in Memcached, takes locality, item size and miss penalty into account. It uses their impacts on service time to determine where a unit of memory space should be (de)allocated. However, PAMA can only predict the data locality in small increments, but pRedis can generate global MRCs. Most previous approaches were implemented on Memcached. Due to the complexity of Redis (e.g. no class supporting, hard to construct MRCs for non-uniform data size), they cannot be directly ported to Redis, so we didn't compared with them. In a recent study, Hyperbolic Caching [16] adopts a priority function that combines frequency, cost, expiration, and other factors to determine a cache replacement victim. We compare Hyperbolic Caching with our approach.

MRC Guided Cache Partitioning Research on cache modeling and MRC construction has been focused on efficient LRU and stack algorithms [27, 28, 31, 32, 38, 39]. Recent approximation algorithms (e.g., CounterStacks [39], SHARDS [38] and AET [27, 28]) output lightweight, continuously-updated MRCs for online modeling and control of LRU caches. A miniature simulation method [37] is proposed for a complex cache using a replacement policy not limited to LRU. For a KV cache based on LRU, EAET extends the AET model to handle different types and sizes of data objects [34]. We use EAET to construct the MRC for pRedis.

In high-throughput storage systems, fast MRC tracking is always beneficial. Several studies use on-line MRC analysis for cache partitioning [36, 42], page size selection [21], and memory management [29, 43]. The memory cache prediction [15] also uses online MRC detection for storage workloads. There are some earlier studies on optimizing memory allocation, e.g. LAMA [25, 26], Dynacache [22], Cliffhanger [23], which use miss/hit rate to optimize memory allocation in Memcached. Another work called mPart [19] is also an MRC-based memory allocation technique, which uses the AET-based MRCs to optimize memory allocation in a multi-tenant key-value store, and outperforms an existing state-of-the-art sharing model, Memshare [24].

8 CONCLUSION

We have presented a systematic design and implementation of pRedis, a penalty and locality aware memory allocation scheme for Redis. pRedis exploits the data locality and miss penalty, in a quantitative manner, to guide the memory allocation in Redis. With the penalty class implemented in Redis, we can effectively manage memory in Redis. pRedis can predict MRC for each penalty class with a 98.8% accuracy and has the ability to adapt the phase change. It outperforms a state-of-the-art penalty aware cache management scheme, HC, by reducing 14 ∼ 52% average response time.

ACKNOWLEDGMENTS

We would like to thank our shepherd Michael Wei and anonymous reviewers for their constructive comments and suggestions. The research is supported in part by the National Key R&D Program of China under Grant No. 2018YFB1003505, and the National Science Foundation of China (Nos. 61472008, 61672053 and U1611461), Shenzhen Key Research Project No. JCYJ20170412150946024, and National Science Foundation CSR1618384.

REFERENCES

[1] Approximated LRU. https://redis.io/topics/lru-cache. Accessed: 2010-06-10.

[2] Memcached Memory Management Blog. https://www.loginradius.com/engineering/memcach-memory-management. Accessed: 2019-06-10.

[3] Memcached Website. http://memcached.org. Accessed: 2019-06-10.

[4] Memtier_benchmark. https://github.com/GarantiaData\/memtier_benchmark. Accessed: 2019-06-10.

[5] MSR Cambridge Traces. http://iotta.snia.org/traces/388. Accessed: 2019-06-10.

[6] MurmurHash. https://en.wikipedia.org/wiki/Murmur-\Hash. Accessed: 2019-06-10.

[7] MySQL Website. https://www.mysql.com. Accessed: 2019-06-10.

[8] MySQLslap. https://tosbourn.com/mysqlslap-a-quickstart-guide/. Accessed: 2019-06-10.

[9] PostgreSQL Website. https://www.postgresql.org/. Accessed: 2019-06-10.

[10] Redis as an LRU cache. http://oldblog.antirez.com/post/redis-as-LRU-cache.html. Accessed: 2019-06-10.

[11] Redis Website. https://redis.io. Accessed: 2019-06-10.

[12] Yahoo! Cloud Serving Benchmark (YCSB). https://github.com/brianfrankcooper/YCSB. Accessed: 2019-06-10.

[13] Zipfian's Law. https://en.wikipedia.org/wiki/Zipf%27s_law. Accessed: 2019-06-10.

[14] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-scale Key-value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '12)*. ACM, New York, NY, USA, 53–64. https://doi.org/10.1145/2254756.2254766

[15] Hjortur Bjornsson, Gregory Chockler, Trausti Saemundsson, and Ymir Vigfusson. 2013. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, Seattle, WA, USA, 1–14.

[16] 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

[17] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (July 1970), 422–426. https://doi.org/10.1145/362686.362692

[18] Jacob Brock, Chencheng Ye, Ding Chen, Yechen Li, Xiaolin Wang, and Yingwei Luo. 2015. Optimal Cache Partition-Sharing. In *International Conference on Parallel Processing, ICPP'15*. 749–758.

[19] 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 (ISMM 2018)*. ACM, New York, NY, USA, 84–95. https://doi.org/10.1145/3210563.3210571

[20] Pei Cao and Sandy Irani. 1997. Cost-aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97)*. USENIX Association, Berkeley, CA, USA, 18–18. http://dl.acm.org/citation.cfm?id=1267279.1267297

[21] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. 2005. Multiple Page Size Modeling and Optimization. 339–349.

[22] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/cidon

[23] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 379–392. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/cidon

[24] 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. https://www.usenix.org/conference/atc17/technical-sessions/presentation/cidon

[25] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX ATC'15*. 57–70.

[26] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. 2017. Optimizing Locality-Aware Memory Management of Key-Value Caches. *IEEE Trans. Comput.* 66, 5 (May 2017), 862–875. https://doi.org/10.1109/TC.2016.2618920

[27] 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)*.

[28] 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

[29] Yul H. Kim, Mark D. Hill, and David A. Wood. 1991. Implementing Stack Simulation for Highly-Associative Memories. *Acm Sigmetrics Performance Evaluation Review* 19, 1, 212–213.

[30] Conglong Li and Alan L. Cox. 2015. GD-Wheel: A Cost-aware Replacement Policy for Key-value Stores. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. ACM, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/2741948.2741956

[31] 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

[32] Qingpeng Niu, James Dinan, Qingda Lu, and P. Sadayappan. 2012. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 1284–1294. https://doi.org/10.1109/IPDPS.2012.117

[33] J. Ou, M. Patton, M. D. Moore, Y. Xu, and S. Jiang. 2015. A Penalty Aware Memory Allocation Scheme for Key-Value Cache. In *2015 44th International Conference on Parallel Processing*. 530–539. https://doi.org/10.1109/ICPP.2015.62

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

[35] J. Shim, P. Scheuermann, and R. Vingralek. 1999. Proxy cache algorithms: Design, implementation, and performance. *Knowledge Data Engineering IEEE Transactions on* 11, 4 (1999), 549–562.

[36] G Edward Suh, Srinivas Devadas, and Larry Rudolph. 2001. Analytical cache models with applications to cache partitioning. In *Proceedings of the 15th international conference on Supercomputing, ICS'01*. 1–12.

[37] 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

[38] 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.

[39] 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

[40] Chencheng Ye, Jacob Brock, Ding Chen, and Hai Jin. 2017. Rochester Elastic Cache Utility (RECU): Unequal Cache Sharing is Good Economics. *International Journal of Parallel Programming* 45, 1 (2017), 30–44.

[41] Neal E. Young. 1991. Competitive Paging and Dual-Guided On-Line Weighted Caching and Matching Algorithms. In *PhD thesis, Princeton University*.

[42] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. 2009. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European conference on Computer systems*. ACM, 89–102.

[43] Pin Zhou, Vivek Pandey, Jagadeesan Sundaresan, Anand Raghuraman, Yuanyuan Zhou, and Sanjeev Kumar. 2004. Dynamic Tracking of Page Miss Ratio Curve for Memory Management. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XI)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/1024393.1024415