PACE: Penalty Aware Cache Modeling with Enhanced AET

Cheng Pan*
Peking University
pancheng@pku.edu.cn

Xiameng Hu
Peking University
huxiameng@gmail.com

Lan Zhou
Peking University
lanzhou@pku.edu.cn

Yingwei Luo*
Peking University
lyw@pku.edu.cn

Xiaolin Wang
Peking University
wxl@pku.edu.cn

Zhenlin Wang
Michigan Technological University
zlwang@mtu.edu

ABSTRACT

Past cache modeling techniques are typically limited to a cache system with a fixed cache line/block size. This limitation is not a problem for a hardware cache where the cache line size is uniform. However, modern in-memory software caches, such as Memcached and Redis, are able to cache varied-size data objects. A software cache supports update and delete operations in addition to only reads and writes for a hardware cache. Moreover, existing cache models often assume that the penalty for each cache miss is identical, which is not true especially for software cache targeting web services, and past cache management policies that aim to improve cache hit rate are no longer sufficient.

We propose a more general cache model that can handle varied cache block sizes, nonuniform miss penalties, and diverse cache operations. In this paper, we first extend a state-of-the-art cache model to accurately predict cache miss ratios for variable cache sizes when object size, updates and deletions are considered. We then apply this model to drive cache management when miss penalty is brought into consideration. Our approach delivers better results than a recent penalty-aware cache management scheme, Hyperbolic Caching, especially when cache budget is tight. Another advantage of our approach is that it provides predictable and controllable cache management on cache space allocation, especially when multiple applications share the cache space.

*Also with Shenzhen Key Lab for Information Centric Networking & Blockchain Technology (ICNLAB), SECE, Peking University, Shenzhen.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

APSys'18, August 27–28, 2018, Jeju, South Korea

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-6006-7/18/08...\$15.00

<https://doi.org/10.1145/3265723.3265736>

KEYWORDS

General cache system modeling, Miss penalty, Controllable and predictable cache management

ACM Reference format:

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 Asia-Pacific Workshop on Systems, Jeju, South Korea, August 27–28, 2018 (APSys'18)*, 8 pages. <https://doi.org/10.1145/3265723.3265736>

1 INTRODUCTION

Caches are ubiquitous in computer systems. Many studies are focused on cache replacement algorithms, such as FIFO, LRU, LFU, and some advanced cache algorithms like ARC [19], LIRS [15] and 2Q [16]. Although these replacement algorithms do not assume a uniform block size, recent cache modeling techniques (e.g., Counter Stacks [24], SHARDS [23] and AET [13, 14]) that model these algorithms depend on the assumption for simplicity.

Caching has been applied to all levels of the memory or storage architecture, especially for in-memory key-value stores. The in-memory cache systems, such as Redis [3] and Memcached [1], often resort to the LRU (or approximated LRU) algorithm to choose which data item to evict. One of the most important differences from a hardware cache is that the data item size in an in-memory cache is not fixed. The size of an object has impact on cache management and performance so it needs to be modeled.

In addition to various block sizes, the operations in a software cache can be more abundant. Besides read and write operations, a software cache system can support updates, which can change object sizes, and delete operations. To accurately model this type of cache, we must consider the impact of these operations.

Last, a modern in-memory cache can store objects from different sources [4]. As a concrete 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. Miss penalty discrepancy [6] will affect the cache replacement policy and cache modeling.

This paper presents *PACE: Penalty Aware Cache Modeling with Enhanced AET*, an extension of a recent cache model, AET [13, 14]. PACE constructs miss ratio curves (MRCs) for a cache of varied block sizes with updates and deletes. We implement our PACE model in Redis 3.2.9, supporting all data structures in Redis with varied object sizes and modeling the MRCs at runtime with low overhead and high accuracy. Besides, miss penalty is added into consideration when applying PACE to drive Redis's data replacement. We compare our Enhanced AET (EAET) model to the original AET model which assumes a uniform object size. Experimental results show that EAET can deliver better accuracy on MRC construction. We further compare PACE with Hyperbolic Caching (HC) [6], a state-of-the-art penalty-aware cache model. We find that PACE yields better performance than HC, especially when the cache space is small. The average miss penalty is reduced by 9.4% in a collection of real-world cloud traces and 2.1% in the MSR traces [2]. At the same time, PACE can accurately model and predict cache penalty, which provides a predictable cache control technique to deal with multiple applications in a shared cache.

2 MODELING AN LRU STACK

In this section, we will first review the AET model for an LRU cache. Then we will discuss the impact of different types of cache access, read, write, update and delete, on an LRU cache.

2.1 AET Modeling

An LRU cache can be viewed as a stack. Data blocks are sorted from top to bottom by their most recent access time. The recently accessed data is brought to the top of the stack. On a miss, the bottom block will be evicted. The AET models an LRU cache using low-cost, effective *reuse time* sampling [13, 14], and generates a miss ratio curve (MRC) that maps from cache sizes to cache miss ratios. A block's (backward) *reuse time* is the time between the current access to this block and its last access. The AET model obtains reuse time distribution through sampling and then converts the distribution to an MRC. This section reviews the key concepts behind this conversion.

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 get evicted. Given the cache size c , and the average travel speed, $v(t)$, at a time t during this process, we have:

$$\int_0^{AET(c)} v(t) dt = c. \quad (1)$$

The solution to this equation yields the average eviction time. Hu et al. show that $v(t)$ is equal to $P(t)$, the probability that a reference has reuse time greater than t . Now Equation 1 turns to:

$$\int_0^{AET(c)} P(t) dt = c. \quad (2)$$

$P(t)$ can be estimated using the sampled reuse time distribution as follows,

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

where N is the length of the sampled sequence and $rt(i)$ is the number of accesses with reuse time i . The logic behind the relationship between $v(t)$ and $P(t)$ sits in that $P(t)$ represents the probability that the *evicting block* moves down one position toward the stack bottom on an access during the eviction process. Specifically, the AET model shows that when block z is accessed, the evicting block d moves down from stack position $m - 1$ to position m if and only if the reuse time of z is greater than the *arrival time*, T_m , the logical time it takes block d to arrive stack position m from the stack top. Equivalently, the probability for d to move down one position is $P(T_m)$, or the speed at T_m is $P(T_m)$ using logical time and logical access rate of one access per time unit. Note that T_c is the time for the evicting block to reach position c , i.e., eviction. On average, T_c is $AET(c)$. The miss ratio $mr(c)$ is indeed the probability that a reuse time is greater than the average eviction time, $AET(c)$:

$$mr(c) = P(AET(c)) \quad (4)$$

In summary, when the reuse time distribution is sampled, Equation 2 finds $AET(c)$ and then Equation 4 gives the miss ratio. In practice, the miss ratio curve, i.e., $mr(c)$ for all c , can be efficiently calculated in linear time.

2.2 Impacts of Various Operations

In order to extend the AET model to a more general scenario, we first discuss the impact of different operations with different object sizes.

Read Operation Access a d -byte object in an LRU cache.

A hit will move the d bytes to the top of the LRU stack.

On a miss, we must bring the missed data from lower storage or other places to the cache.

Delete Operation Delete a d -byte object from an LRU cache. The effect is that all the data in the LRU stack that reside below the deleted object will move d bytes

toward the LRU stack top. If the object being deleted is not present in cache, there will be no data movement.

Write and Update Operation Write or update a d -byte value with an address/key. If the address/key is not present in cache, the write/update operation will result in an insertion to the stack top such that every data object moves d bytes towards the LRU stack bottom. Otherwise, the update can change the size of an object. Assume that the update changes a d_1 -byte item to d_2 bytes. If d_1 is larger than d_2 , we can consider that d_2 bytes are reused and $d_1 - d_2$ bytes are deleted. If d_1 is equal or smaller than d_2 , d_1 bytes in the original data will be reused and the additional $d_2 - d_1$ bytes are treated as a new write (cold miss).

3 ENHANCED AET MODELING

In this section, we present how to enhance the original AET to model a more general cache system. In our model, we logically decompose each operation into byte-level minimum granularity operations. For instance, a read operation to a d -byte object is decomposed as d read operations to each individual byte of the object. Assume that there is a byte, b , residing in the cache. If the next byte operation is a read, write or update to any byte that has a reuse time larger than the arrival time of b (to one-byte position below), b will move one byte towards the LRU stack bottom because the accessed byte is moved to the stack top. For deletion, we call the (reuse) time between the previous access to an object and its current deletion the *delete time*. If the next byte operation is a delete with a delete time smaller than t , this byte of data will be promoted one byte toward the LRU stack top because the deleted byte was located at a higher stack position.

Now the reuse time distribution is modeled at byte level. Assume that we have the reuse time distribution of all the read, write and update operations to any byte as well as the delete time distribution of all deletions. Assume $fa(i)$ presents the probability that a byte is accessed (read, written or updated) with a reuse time i . $fd(i)$ presents the probability that a byte is deleted with a delete time i . For each byte operation, the probability for a byte of data with arrival time t to move one byte towards the stack bottom is $P(t)$:

$$P(t) = \sum_{i=t+1}^{\infty} fa(i) - \sum_{i=0}^{t-1} fd(i) \quad (5)$$

Equation 5 extends Equation 3 by considering possible upward data movement due to deletions. The movement of a data block can be bi-directional: 1) An downward move, caused by a read, write or update operation; 2) An upward move, caused by a delete operation. $P(t)$ can be negative which indicates an upward movement. The overall speed

of movement is a superposition of the velocities in both directions, $v_{overall} = v_{downward} - v_{upward}$. Here, $v_{downward} = \sum_{i=t+1}^{\infty} fa(i)$ is the same as Equation 3, and $v_{upward} = \sum_{i=0}^{t-1} fd(i)$ is caused by delete operations. 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, Equation 2 can be applied to find the average eviction time and Equation 4 derives the miss ratio curve.

Here, we need to describe the calculation of $fa(i)$ and $fd(i)$. Splitting the original operation into byte-level consecutive operations to calculate the probability is too costly when the average object size is big. In fact, these consecutive byte operations have the same reuse time. Let $access_rth[i]$ record the number of bytes with reuse time i over operation read, write and update, $access_rth[\infty]$ be the number of cold miss bytes, and $delete_rth[i]$ be the number of bytes with delete time i . We assume that the reuse time or delete time of the following four operations is t ($t = \infty$ represents the first touch of an object), and each operation will be processed as described below:

read operation Read an object in cache with size s , ($s = 0$ if the object is not in cache).

If ($s = 0$) then $access_rth[\infty] += object_original_size$;
else $access_rth[t] += s$;

write operation Write an object with size s ,
 $access_rth[t] += s$;

delete operation Delete an object with size s .
 $delete_rth[t] += s$;

update operation Update an object from size s' to size s , ($s' = 0$ if the object is not in cache).

If ($s' = 0$) then $access_rth[\infty] += s$;
else if ($s > s'$) then begin $access_rth[t] += s'$;
 $access_rth[\infty] += s - s'$; end;
else if ($s \leq s'$) then begin $access_rth[t] += s$;
 $delete_rth[t] += s' - s$; end;

With the reuse time distribution of access and delete, we can easily find fa and fd :

$$fa(t) = \frac{access_rth[t]}{N} \quad (6)$$

$$fd(t) = \frac{delete_rth[t]}{N} \quad (7)$$

Finally, we explain how to construct MRCs. In practice, read misses affect a system's performance the most so we focus on constructing read miss ratio curve. Our model decomposes operations into byte-level. However, general read miss ratio is better defined at object level. We adopt *reuse time distribution of (read) requested object* to model a general cache system – it is the probability that a read object's reuse time is larger than $AET(c)$.

Figure 1: Any Byte Miss leads to Whole Object Miss

We use Figure 1 to illustrate the relationship between object movement and byte movement. In our model, all requests enter the LRU stack byte by byte, any byte's reuse time is identical and equals to the object's reuse time. We can see after writing object *B*, the first (lowest-order) byte of object *A* is evicted, and if any byte of an object is evicted, the whole object is evicted. So we can measure read miss ratio by calculating the reuse time distribution of the requested object.

Assume that $request_rth[i]$ is the reuse time distribution for all read operations, which is a count of requested object in all read operations with reuse time i . According to the AET model, we know that when the reuse time of a byte is greater than $AET(c)$, this byte will be a cache miss, so as the whole object. With $AET(c)$ calculated in Equation 2, the read miss ratio of this cache is:

$$read\ miss\ ratio = \frac{\sum_{i=AET(c)+1}^{\infty} request_rth[i]}{\sum_{i=0}^{\infty} request_rth[i]} \quad (8)$$

In Equation 8, the denominator is the number of all read operations, and the numerator is the number of requested objects whose reuse times are greater than $AET(c)$. The overall read miss ratio can be calculated along with AET integration, and other operations affect the read miss ratio by contributing to the calculation of $AET(c)$ using Equation 2.

4 PENALTY AWARE CACHE MANAGEMENT

Because of the complexity of the back-end structure of the cache system, the cost (or penalty) of retrieving different data after a miss is different. The cache miss penalty should be taken into consideration for cache optimization.

The locality of memory access and the penalty of data are two important aspects that affect system performance. We can measure the locality of access through the Enhanced AET (EAET) model, at the same time, we also hope to combine the locality measure and penalty management to improve the overall performance of the cache system, which is PACE's target. In our penalty aware cache management, all items

have an associated cost that reflects the penalty for a miss on the item. Through locality measurements, we can calculate the number of cache misses under different cache sizes, and further calculate the miss penalty. The final goal is to minimize the total penalty of all misses.

Figure 2: The principle of penalty aware cache management

In Figure 2, we illustrate the principle of our PACE model. By partitioning the LRU stack into several classes, we can place data items of different penalty into different classes. The items in the same class have similar miss penalties. The miss ratio for each class can be measured by EAET, and the overall performance of the whole cache system can be improved by a proper class allocation scheme, which will be discussed in Section 4.2.

4.1 Impacts of Miss Penalty on MRC

The original AET Model assumes a cache miss must bring the missed data from lower storage immediately. However, real-world cache systems usually have non-blocking mechanism to improve parallelism and service time. When a miss occurs, we cannot bring the missed data back immediately. Non-blocking cache will continue its service, so the data access sequence to the cache is not the same as the data arrival sequence. Various miss penalties will further increase the non-determinism of access ordering. Although changes of access ordering due to miss penalty can affect MRC, we find that the impact is indeed minimal.

We use the MSR traces (see Section 5, [2]) to evaluate the impact of the miss penalty on the accuracy of the MRC construction. Here, we estimate the miss penalty as the time gap between the miss of a GET request and the SET of the same key immediately following the GET [20]. In Figure 3, we pick 4 out of 13 MSR traces to illustrate the impacts of miss penalty on MRC accuracy. We simulated the miss ratio of an LRU stack at different cache sizes with each data item having a miss penalty, and construct the MRC using the AET model without considering the impacts of the miss penalty.

Figure 3: Impacts of Miss Penalty on MRC Accuracy

Results show that the MRCs are almost identical with and without modeling miss penalty. This means we can reliably model cache by considering the ordering of the incoming requests only.

4.2 Memory Partitioning Guided by MRC

We propose to optimize the cache performance with penalties through cache partitioning. From our EAET model, we can accurately predict the cache miss ratio for any cache size on a general cache system. By grouping data items into different penalty classes, we can calculate the miss ratio of each penalty class at different cache sizes. If we allocate penalty class i with M_i memory, then this class's overall miss penalty MP_i can be estimated as:

$$MP_i = mr_i(M_i) * p_i * N_i, \quad (9)$$

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

$$\min \sum_{i=1}^C MP_i = \min \sum_{i=1}^C mr_i(M_i) * p_i * N_i \quad (10)$$

$$s.t. \sum_{i=1}^C M_i = M, \quad (11)$$

where C is the number of penalty classes, M is total memory we can allocate. The optimal solution can be reached through dynamic programming (see Algorithm 1).

Obviously, real-world workloads change their access patterns over time. In order to achieve high performance, we need to partition the memory periodically. A workload can be divided into a series of fixed-size time windows. In a

time window, we sample data accesses within the window to build the MRC, calculate the best allocation scheme to achieve minimal total miss penalty, and finally apply this allocation scheme to next time window. The size of time window can be set by the user or empirically.

Algorithm 1 Dynamic Programming for Penalty Class Allocation

```

1: for  $i \leftarrow 1..C$  do
2:   for  $j \leftarrow i + 1..M$  do
3:      $f[i][j] \leftarrow \infty$ 
4:     for  $k \leftarrow i..j$  do
5:        $f[i][j] \leftarrow \min(f[i][j], f[i-1][k] + MR_i[j-k] * p_i * N_i)$ 
6:     end for
7:   end for
8: end for

```

5 EVALUATION

Our evaluation focuses on answering the following questions:

- (1) Can our EAET model accurately predict miss ratio curves with non-fixed-size data items and various types of operations?
- (2) How does the performance of our penalty-aware partitioning algorithm compare to the existing penalty-aware algorithms?

We implement PACE using a simulator and inside Redis, and run real-world traces. The evaluation has been conducted on a Ubuntu 16.04 servers located on a single rack with Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz and 200GB of memory.

The workloads in our experiments include the MSR Cambridge traces [2] and a collection of seven Redis traces from a cloud computing service provider. The MSR traces represent one week of block I/O traces from the Microsoft Research Cambridge Enterprise servers. In the evaluation, the raw MSR Traces data are transformed to emulate in-memory KV-store operations with SET and GET. The Redis traces were obtained from a set of Redis servers used for electronic 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. Table 1 shows the characteristics of these traces.

5.1 MRC Accuracy of Enhanced AET

First of all, we need to verify that our Enhanced AET model can accurately predict the miss ratio on varied object sizes.

Table 1: Redis Trace Details

Trace No.	1	2	3	4
Trace Length	59616194	18687961	194354795	29900738
Trace Period	afternoon	midnight	daytime	midnight
Trace No.	5	6	7	
Trace Length	71014252	73400682	89939618	
Trace Period	morning	afternoon	midnight	

We compare it with the original AET and LRU. The original AET, denoted as *Uni AET*, assumes that the size of each accessed data object is uniform. We use the sampled average object size to drive Uni AET. Similarly, *Uni LRU* models the LRU policy assuming uniform object size. In order to let the original AET model non-fixed data size, we decompose each operation into minimum granularity (block in the MSR traces) operations, denoted as *Per-block AET*, or *PerB AET*. PerB AET simulates a request using multiple consecutive requests on single blocks. Finally, we implement an LRU simulator which supports various object sizes in cache replacement, denoted as *Var LRU* (as the upper bound).

Figure 4: MRC Accuracy in MSR Traces

We transform an MSR trace to a KV-store trace by treating *addr* as the key, *size* as the size of value. In Figure 4, we pick 4 out of 13 MSR traces to demonstrate the accuracy. The results for others are similar.

Obviously, we can find that using average data size to construct MRC is not accurate, shown as Uni LRU and Uni AET in Figure 4. Because of the size distribution of the data and the frequency distribution of visits are not uniform, using the average size will cause the MRC curve to shift.

Figure 5: MRC Accuracy in Redis Traces

At the same time, we can see that Enhanced AET achieves almost the same accuracy as PerB AET and performs close to Var LRU, which serves as the upper bound. However, Enhanced AET runs much faster than PerB AET as shown in Table 2. Here, the time is the overall processing time excluding input/output time.

Table 2: EAET¹ vs PerB AET on Time Consumption

Traces	hm	mds	prn	ts	web	stg	proj
EAET	16.3s	12.6s	92.0s	8.7s	67.0s	31.5s	3646.6s
PerB ²	144.9s	776.1s	1450.1s	67.3s	1375.4s	675.1s	>5hrs
Speedup	8.9x	62.0x	15.8x	7.7x	20.5x	21.4x	>4.9x

Traces	prxy	rsrch	src1	src2	usr	wdev
EAET	1418.0s	6.4s	1170.6s	31.1s	977.2s	9.9s
PerB	9500.4s	32.8s	>5hrs	643.5s	>5hrs	62.1s
Speedup	8.3x	5.1x	>15.4x	20.7x	>18.4x	6.3x

¹ EAET means Enhanced AET Model

² PerB means Per-Block AET Model

We also implement Enhanced AET in Redis 3.2.9 to track MRCs and evaluate the accuracy using the seven Redis traces. Moreover, we compare the predicted miss ratio against the real miss ratios through Redis statistics. Figure 5 show the results for traces No. 1 and No. 5 and the results for the remaining 5 traces are similar. The *Redis Statistics* miss ratio curves are obtained by running the Redis instance multiple times under different maximum-memory configurations, and the actual miss ratios are fetched from Redis statistics info. We can observe that the miss ratio curve can be very accurately predicted by our Enhanced AET model.

5.2 Penalty Management

Hyperbolic Caching (HC) [6] is a recent study on penalty-aware cache. Inspired by Redis's random sampling approach and the LFU policy for replacement, HC incorporates both to derive a cost model for cache replacement. HC shows its effectiveness in managing a penalty-aware cache, so we reproduce HC and compare it with our PACE model.

To manage the miss penalty, we measure the time interval between the miss of a GET request and the SET of the same key immediately following the GET request. We sort and group these miss penalties to 10 penalty classes according to 10 fixed ranges. We set the penalty range of each class following a power of 2 increment. In practice, we assume a system can set the penalty class based on the type of a KV object. We run the MSR traces and the Redis traces on PACE and HC, set the cache size from 10%, 40%, 70% to 100% of the data set size, and then compare the final total miss penalty under different cache sizes.

Figure 6: Miss penalty in Redis Traces

In Figure 6 and Figure 7, we can see that PACE is able to reduce the total penalty when compared to HC, especially when the cache size is relative small (e.g. 10% of data set size). The average total miss penalty is reduced by 9.4% in the Redis traces and 2.1% in the MSR traces. The maximum total penalty is reduced by 19.2% in the Redis traces and 7.9% in the MSR traces. In addition to being able to obtain a lower miss penalty, another advantage of PACE over HC is that the miss penalty is predictable and controllable under PACE. PACE models and predicts the total miss penalty for the current configuration, but HC takes a greedy approach without modeling the global effect. When multiple applications are running in a shared cache, the ability to predict the performance of each application is often desirable to deliver reasonable Quality of Service. Our model can adjust the performance of individual applications when sharing memory.

Figure 7: Miss penalty in MSR Traces

6 RELATED WORK

MRC profiling techniques are widely used in different applications. Several studies use on-line MRC analysis for cache partitioning [22, 26], page size selection [9], and memory management [17, 27]. The memory cache prediction [5] also uses on-line MRC detection for storage workloads. In high-throughput storage systems, fast MRC tracking is always beneficial. One earlier study on optimizing memory allocation is LAMA [11, 12], which uses footprint-based MRC to optimize memory allocation in Memcached, an in-memory key-value store. LAMA shows that MRC-based optimization is superior to the Facebook and Twitter approaches in steady-state performance. Another work called mPart [7] 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 [10].

Cost-aware caching is a recent research hot spot. GreedyDual [25] attempts to incorporate cost into LRU, requiring a re-design. Cao and Irani [8] implement GreedyDual using priority queues for size-aware caching in web proxies. GDWheel [18] implements GreedyDual in Memcached using a more efficient wheel data structure. PAMA [20] is a novel work which 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. Other cost-aware strategies have incorporated properties such as freshness (e.g., [21]). In contrast to these approaches, Hyperbolic Caching [6] proposes a priority function that can adopt frequency, cost, expiration, or other factors.

7 CONCLUSION

We have presented the design and implementation of PACE, a general cache model and management method. The evaluation in Redis shows that we can accurately construct MRCs on read, write, update and delete operations with non-fixed data sizes. Minimizing total miss penalty by partitioning among penalty classes helps us to create a controllable and predictable cache controlling method. In addition, this method delivers better performance than a recent penalty-aware cache management scheme, Hyperbolic Caching.

8 ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their constructive comments and suggestions. The research is supported in part by the National Science Foundation of China (No. 61472008, 61672053 and U1611461), Shenzhen Key Research Project No. JCYJ20170412150946024, National Science Foundation CSR1618384 and the National Key R&D Program of China under Grant No. 2018YFB1003604.

REFERENCES

- [1] 2018. Memcached Website. <https://memcached.org/>. (2018). Accessed: 2018-01-20.
- [2] 2018. MSR Cambridge Traces. <http://iotta.snia.org/traces/388>. (2018). Accessed: 2018-01-20.
- [3] 2018. Redis Website. <https://redis.io/>. (2018). Accessed: 2018-01-10.
- [4] 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>
- [5] 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, 59.
- [6] 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>
- [7] 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>
- [8] 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>
- [9] Calin Cascaval, Evelyn Duesterwald, Peter F. Sweeney, and Robert W. Wisniewski. 2005. Multiple Page Size Modeling and Optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*. 339–349.
- [10] 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>
- [11] 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*.
- [12] 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>
- [13] 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)*.
- [14] 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>
- [15] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '02)*. ACM, New York, NY, USA, 31–42. <https://doi.org/10.1145/511334.511340>
- [16] Theodore Johnson and Dennis Shasha. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 439–450. <http://dl.acm.org/citation.cfm?id=645920.672996>
- [17] Yul H. Kim, Mark D. Hill, and David A. Wood. 1991. Implementing Stack Simulation for Highly-Associative Memories. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*. 212–213.
- [18] 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>
- [19] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*, Vol. 3. 115–130.
- [20] 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>
- [21] Junho Shim, Peter Scheuermann, and Radek Vingralek. 1999. Proxy cache algorithms: design, implementation, and performance. *IEEE Transactions on Knowledge and Data Engineering* 11, 4 (7 1999), 549–562. <https://doi.org/10.1109/69.790804>
- [22] 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*. ACM, 1–12.
- [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] Neal E. Young. 1991. Competitive Paging and Dual-Guided On-Line Weighted Caching and Matching Algorithms. In *PhD thesis, Princeton University*.
- [26] 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.
- [27] 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>