LEA: A Lazy Eviction Algorithm for SSD Cache in Cloud Block Storage

Ke Zhou[†], Yu Zhang[†], Ping Huang[§], Hua Wang[†*], Yongguang Ji[‡], Bin Cheng[‡], Ying Liu[‡]
[†]*Wuhan National Laboratory for Optoelectronics (Huazhong University of Science and Technology),*
[†]*Key Laboratory of Information Storage System, Intelligent Cloud Storage Joint Research center of HUST and Tencent*
[§]*Temple University,* [‡]*Tencent Technology (Shenzhen) Co., Ltd.*
Email:{k.zhou, yuzhang2016, hwang}@hust.edu.cn, templestorager@temple.edu
{raidmanji, bencheng, burtliu, reganxie}@tencent.com
[*]*Corresponding author: Hua Wang (hwang@hust.edu.cn)*

Abstract—**Solid State Drives (SSDs) are popularly used for caching in large scale cloud storage systems nowadays. Traditionally, most cache algorithms make replacement at each miss when cache space is full. However, we observe that in a typical Cloud Block Storage (CBS), there is a great percentage of blocks with large reuse distances, which would result in large number of blocks being evicted out of the cache before they ever have a chance to be referenced while they are cached, significantly jeopardizing the cache efficiency. In this paper, we propose LEA, Lazy Eviction cache Algorithm, for cloud block storage to efficiently remedy the cache inefficiencies caused by cache blocks with large reuse distances. Specifically, LEA uses two lists, Lazy Eviction List (LEL) and Block Identity List (BIL). When a cache miss happens, if the candidate evicted-block has not resided in cache for longer than its reuse distance, LEA inserts the missed block identity into BIL. Otherwise, it inserts the missed block entry into LEL. We have evaluated LEA by using IO traces collected from Tencent, one of the largest network service providers in the world, and several open source traces. Experimental results show that LEA not only outperforms most of the state-of-the-art cache algorithms in hit ratio, but also reduces the number of SSD writes greatly.**

Index Terms—**Cloud Block Storage, Cache Algorithm, SSD, Reuse Distance**

I. Introduction

Solid State Drives (SSDs), used as a caching layer, have been widely deployed in cloud storages because of their great random read and write performance. However, designing the most appropriate caching algorithm largely determines the performance of the storage system.

Exisiting advanced cache algorithms make replacement on each miss when the cache space is full, which is mainly based on the principle of locality. However, we find that in a typical Cloud Block Storage (CBS), there is a great percentage of blocks with large reuse distances, which implies a lot of blocks will be re-referenced after a long time. If simple replacement is made upon each miss, newly accessed blocks will pollute the cache without bringing any hits, reducing cache efficiency. In addition, we also find a similar phenomena in other scenarios using public IO traces collected from other data centers.

This work is supported in part by the National Natural Science Foundation of China under Grant No.61502189, and the National Key Research and Development Program of China under Grant No.2016YFB0800402.

In this paper, we propose a novel cache algorithm, LEA, for CBS and other similar scenarios. LEA does not make replacement by default when there is a cache miss except when certain conditions are met. The conditions are defined by two aspects, i.e., the frequency of the newly accessed blocks and the value of the candidate evicted-block in the cache. Here the "value" is defined as the importance or usefulness of this block for the cache (the same below). By doing this, the residence time of blocks in cache can be largely extended. More importantly, the number of writes to SSD can be largely reduced, which can extend the lifetime of SSD. The contributions of this study are as follows.

- We propose a novel cache algorithm LEA which is suitable for cloud block storages and other similar scenarios. LEA not only outperforms the state-of-the-art cache algorithms in hit ratio, but also reduces the number of writes to the SSD cache significantly.
- LEA is implemented with low complexity O(1) by two simple lists (i.e., Lazy Eviction List and Block Identity List).
- We have experimentally demonstrated the efficiency of LEA using IO traces collected from Tencent CBS as well as several public IO traces.

II. Background and Motivation

A. Reuse Distance of Cloud Systems

In a cache system, data reuse distance has a great impact on cache hit ratio. Generally, different systems have different characteristics of reuse distance. For example, in the context of traditional on-chip cache, the average reuse distance could be much shorter than that of a cloud storage system. For the convenience of exposition, we first introduce the definition of reuse distance.

The reuse distance of a block is defined as the count of unique blocks between two consecutive references to the same block [10]. It also has been called as inter-reference recency (IRR) [2]. For example, assuming a block sequence is 1-2-4-5-4-3-3-2, the reuse distance of Block 2 is 3, because the unique block set between two consecutive references to Block 2 is {3,4,5}. There are also many studies (e.g., [15] and [16]) that define the reuse distance as the count of blocks between two consecutive references to the same block. And according to this definition, the reuse distance of Block 2 is 5. In this study, we adopt the second definition of reuse distance for the sake of simplicity.

We find that the block reuse distance increases significantly in CBS. As shown in Fig. 1, the solid line represents the complementary cumulative distribution function of block resuse distance for CBS accesses. We can get a deeper understanding of the diagram by looking at point A as marked in the Fig. 1. Point A(32GB, 50%) means that for CBS accesses, there are 50% of the blocks whose reuse distances are larger than 32GB. So if we use a cache algorithm like LRU, 50% of the reused blocks would not hit when the cache size is 32GB. Equally said, it may require hundreds of gigabytes of cache size to make the vast majority of reused blocks to be hit, which is cost-wise unacceptable.

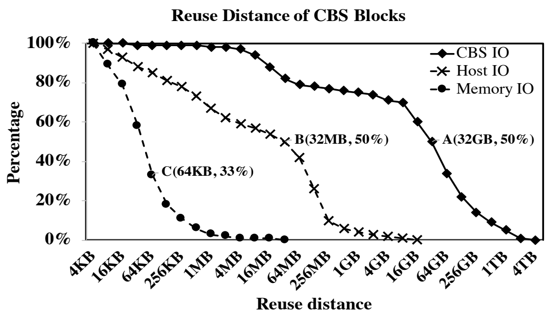

Fig. 1. Reuse distance of CBS blocks

The phenomenon in CBS is somewhat similar to the loop pattern, but they are different. For a loop pattern, most of the block reuse distances are very large. Algorithms like MRU can work efficiently for this kind of loop accesses. However, for CBS accesses, there are still considerable blocks with small reuse distances, which are not well taken into consideration by algorithms like MRU.

B. *Inefficiencies of Existing Algorithms*

Cache algorithms have been developed for decades, from traditional on-chip cache to cloud cache. However, it is inappropriate to simply use conventional cache algorithms to manage SSD cache for CBS due to the following two reasons.

Firstly, most conventional cache algorithms (e.g., LRU, MRU, LFU, CLOCK, FIFO and so on) will make replacement on each miss when the cache space is full. These algorithms can work well for on-chip cache because of its strong temporal locality. However, we find that in CBS, the block reuse distance increases significantly, which means many blocks will be re-referenced only after a long time. If we naively make replacement upon each miss, newly accessed blocks will pollute the cache but might never be hit in the future. Algorithms such as LRU-K [5], EELRU [7], 2Q [8], MQ [9], ARC [1], LIRS [2] and FRD [10] all have ghost lists to help recording the history of IO sequences, which can alleviate the above problem to some extent. However, when the reuse distance is larger than the size of ghost lists, these algorithms will suffer from the similar problem. By contrast, LEA algorithm does not make replacement by default, which can largely extend the residence time of blocks in the cache, alleviating the problem.

In addition, an SSD has limited write endurance. The number of writes to an SSD is an important indicator for SSD cache algorithms [11] [12] [13] [14]. There are many algorithms proposed to reduce the number of writes to the SSD and extend the lifetime of SSD, such as LARC [15] and S-RAC [16]. They typically use a filter list to prevent blocks from entering the cache unless when blocks have been accessed for twice (for LARC) or more times (for S-RAC). In doing this, they can eliminate a lot of useless write traffic to SSD. However, these algorithms only envaluate the value of the missed blocks and may sacrifice some hit ratio at the same time. LEA differs from these algorithms and it also considers the value of the candidate evicted-block in the cache.

Studies like [17] are also designed to reduce the number of writes to SSDs, but they don't apply to our scenario: First, they are applied to write-only caches. Second, they require application level hints (such as the application type of each process, etc.) which can't be got at CBS for the consideration of tenants' privacy.

III. Design and Implementation

A. *LEA Algorithm*

LEA is implemented by using two lists, Lazy Eviction List (LEL) and Block Identity List (BIL), both of which are of equal list length. Each list has two end points, i.e., Insertion Point (IP) and Eviction Point (EP). LEL is the cache list used to store block entries, while BIL is a ghost list used to store identities of blocks. An identity only includes the volume number and address information (offset) of a block to distinguish it from others. And an entry contains the block data as well as some more block information such as *last_access*, *reuse_distance*, *age*, *flag*, etc. This block information is used to evaluate the value of the entry block. The *age* indicates the time[1] difference between the current time and the last accessed time (*last_access*) of the block. If the *age* of a block is smaller than it's *reuse_distance*, it indicates the block will be valuable for the cache. *flag* is also used to judge the block value, which increases by 1 when the block is hit in the LEL list and decreases to the half when the block is the candidate evicted-block but is not evicted due to the lazy replacement. If the *flag* is greater than 0, it indicates the block is valuable for the cache.

When block X is accessed for the first time, it either inserts the identity of X to BIL if the lazy conditions are met, or inserts the entry of X to LEL. When the re-reference to block X happens, the processing is similar but this lazy condition is much more difficult to satisfy than the previous one because

[1]The time here is logical time which increases by 1 when a new block request comes.

block X has been accessed more than once and it should have a higher probability of entering the LEL list.

Tab. I shows the pseudo-code of lazy conditions. LAZY_1() executes when it is the first access of X, while LAZY_2() executes when X is re-referenced. These two conditions are used to test whether LEL.EP is really useless for the cache now. It is easy to find that the condition of LAZY_2() is much more difficult to be met than that of LAZY_1(). This is because in the situation of calling LAZY_2(), block X has been referenced at least twice in BIL and proved to be useful. So block X should have more chance to be inserted into LEL.

There are two points worthy be noted. Firstly, the variable *LEL.EP.age* is the difference between variable *cnt* and *LEL.EP.last_access*. And this variable means how long LEL.EP has been in the cache. Secondly, there is a lazy parmeter K in LAZY_2(), which is used to indicate the degree of laziness. A larger K means being lazier. From the experiments, we found that for most workloads, the best value of K is 1.

TABLE I
THE PSEUDO-CODE OF LAZY CONDITIONS

```
Pseudo-Code 2: Subroutine
LAZY_1()
if LEL.EP.flag > 0 then
  return TRUE
else
  return FALSE


LAZY_2()
if LEL.EP.flag > 0 and ( LEL.EP.age < LEL.EP.last_reusedis
  × LEL.EP.flag × K ) then
  return TRUE
else
  return FALSE
```

Tab. II shows the pseudo-code of our LEA algorithm. In the code, PARA is another lazy parameter. Larger PARA means being lazier. From comprehensive experiments, we find that for most workloads, the best value of PARA is 2.

IV. Experimental Results

We have evaluated the performance of LEA algorithm using a home-made trace-driven simulator BlockCacheSim [18], which we have made publicly available. We have compared LEA with other four algorithms, including LRU, ARC, LARC as well as OPT [19]. Among them, OPT can provide for us the theoretical upper limit of a cache algorithm.

A. Traces

To demonstrate the efficiency of LEA, we have collected IO traces from Tencent CBS by using a proxy server near the storage nodes where the cache is desired to be employed. Furthermore, LEA is also applicable to other scenarios with relatively large block-reuse-distance. To demonstrate this, we also tested LEA via using public IO traces collected from other data centers [20] [21].

TABLE II
THE PSEUDO-CODE OF LEA

```
Pseudo-Code 1: LEA Algorithm
INITIALIZATION: Set cnt = 0
Assume access to block X
cnt++
if LEL Miss then
  if BIL Miss then
    if LAZY_1( ) is TRUE then                      CASE 1
      LEL.EP.flag >> 1
      move LEL.EP to LEL.IP
      Evict BIL.EP
      Insert X.identity into BIL.IP
    else                                           CASE 2
      Evict LEL.EP
      X.flag = PARA
      X.last_access = cnt
      Insert X into LEL.IP
    end if
  else
    if LAZY_2( ) is TRUE then                      CASE 3
      LEL.EP.flag >> 1
      move LEL.EP to LEL.IP
      move X.identity to BIL.IP
    else                                           CASE 4
      X.flag = PARA
      X.last_access = cnt
      move LEL.EP.identity to BIL.IP
      Evict LEL.IP
      move X to LEL.IP
  end if
else                                               CASE 5
      X.flag++
      X.last_reusedis = cnt - X.last_access
end if
```

tencent_0, *tencent_1*, *tencent_2*, *tencent_3*, *tencent_4* were collected from Tencent CBS. *tencent_0* and *tencent_1* are write-intensive. *tencent_2* and *tencent_3* are read-intensive and read-write balanced, respectively. *tencent_4* is an one week-long trace which is tested to evaluate the consistence and flexibility of LEA algorithm. The amount of original data is specifically large, reaching the scale of petabyte magnitude. So we use a sampling method to reduce the amount of test data.

Other traces are from open source repositories. *websearch* is from a web search engine [20]. *adsds* is a trace collected over a period of 24 hours for display ads platform data server [21]. *online* is from a course management system of FIU [21]. And *stg_0* is for web staging [21].

B. Results

In this section, we give a detail description of the experimental results. The cache sizes of *tencent_0*, *tencent_1*, *tencent_2* and *tencent_3* are set from 0.8GB to 4GB [1]. The cache sizes of *websearch*, *adsds*, *online* and *stg* are set from 3GB to 6.2GB, 1.5GB to 2.3GB, 78MB to 390MB and 60MB

[1]This is the result of sampled IO traces, and the cache size should be set in range from hundreds to thousands of GB in practicle. And we set cache sizes as these values to ensure the hit ratio varies from relatively small to large.

to 140MB, respectively. Fig. 2 shows the results of hit ratio, while Fig. 3 shows the results of the number of writes. All the results are averages of multiple tests with different cache sizes. It should be noted that the numbers in Fig. 3 are normalized values. The number of writes of other algorithms is normalized to that of LARC algorithm.

1) Hit ratio: As can be seen from Fig. 2, LEA always achieves the highest hit ratio compared to other algorithms. On average, the hit ratio of LEA (45.13%) exceeds LRU (36.45%) by 23.80%, LARC (39.00%) by 15.70% and ARC (41.15%) by 9.66%.

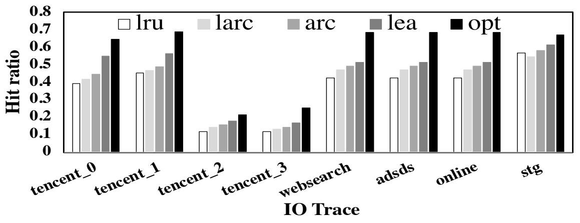

Fig. 2. The comparsion of hit ratio

2) The number of writes to SSD: As for the number of writes to SSD, LEA is of 62.85% lower than LRU and of 61.85% lower than ARC. LARC has the lowest write traffics, which is of 34.65% lower than LEA. This is achieved at the expense of hit ratio, which is not worth the candle because hit ratio has a greater effect on the performance of storage systems.

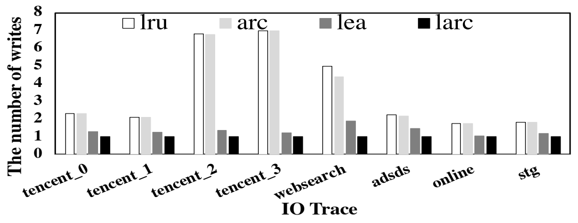

Fig. 3. The comparsion of the number of writes to SSD

3) The Consistence and Flexibility of LEA: As can be seen from Fig. 4 (a) and Fig. 4 (b), LEA always has the highest hit ratio compared to other algorithms. On average, the hit ratio of LEA (47.63%) exceeds LRU (40.90%) by 16.45%, LARC (42.39%) by 12.32% and ARC (44.48%) by 7.08%. There is also a good performance for LEA as for the number of writes to SSD. On average, the number of writes to SSD when using LEA is of 50.64% lower than LRU and 50.21% lower than ARC.

V. Conclusion

SSD-based cache have been widely deployed to improve the random read and write performance of CBS. In this paper we propose a novel algorithm for CBS named Lazy Eviction cache Algorithm (LEA). LEA can largely extend the time that data stay in the cache. The existing data in the cache can be hit as much as possible. More importantly, when applied to

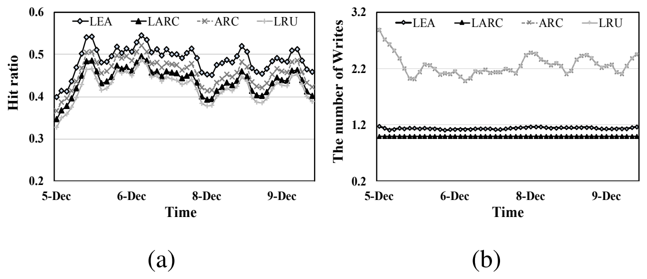

Fig. 4. The hit ratio(a) and the number of writes to SSD(b) by using *tencent_4*

SSD-based cache, LEA can significantly reduce the number of writes to the SSD devices, which results in extended lifetime of the devices. LEA has very low overhead of complexity of O(1). Experimental results show that LEA can not only outperforme the state-of-the-art cache replacement algorithms for hit ratio, but can also reduce the number of writes to SSD significantly.

References

[1] N. Megiddo et al., "ARC: A SelfTuning, Low Overhead Replacement Cache," in Proceedings of FAST '03.

[2] S. Jiang et al., "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance," in Proceedings of SIGMETRICS, 2002.

[3] D. Lee et al., "LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," in IEEE transactions on Computers, 2001, (12), pp. 1352-1361.

[4] J. T. Robinson et al., "Data cache management using frequency-based replacement," ACM, 1990.

[5] E. J. O'Neil et al., "An optimality proof of the LRU-K page replacement algorithm," in Proceedings of SIGMOD, 1993.

[6] R. Karedla et al., "Caching Strategies to Improve Disk System Performance," Computer, 27(3), 3846. https://doi.org/10.1109/2.268884.

[7] Y. Smaragdakis et al., "The EELRU adaptive replacement algorithm," in Performance Evaluation, 2003, 53, (2), pp. 93-123.

[8] T. Johnson et al., "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in Proceedings of VLDB, 1994.

[9] Y. Zhou et al.,"The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in Proceedings of MSST, 2001.

[10] S. Park et al., "FRD : A Filtering based Buffer Cache Algorithm that Considers both Frequency and Reuse Distance," in Proceedings of MSST '17.

[11] P. Huang, et al., "FlexECC: Partially Relaxing ECC of MLC SSD for Better Cache Performance," in Proceedings of ATC'14.

[12] P. Huang, et al., "An aggressive worn-out flash block management scheme to alleviate SSD performance degradation," in Proceedings of EuroSys'14.

[13] K. Zhou, et al., "Demystifying Cache Policies for Photo Stores at Scale: A Tencent Case Study," in Proceedings of ICS'18.

[14] K. Zhou, et al. "LX-SSD: Enhancing the Lifespan of NAND Flash-based Memory via Recycling Invalid Pages," in Proceedings of MSST'17.

[15] S. Huang et al., "Improving flash-based disk cache with Lazy Adaptive Replacement.," in Proceedings of MSST'13.

[16] Y. Ni et al., "S-RAC: SSD Friendly Caching for Data Center Workloads," in Proceedings of SYSTOR'16.

[17] S. Kim et al., "Request-Oriented Durable Write Caching for Application Performance," in Proceedings of ATC'15.

[18] BlockCacheSim. [Online]. Available: https://github.com/zydirtyfish/BlockCacheSim

[19] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," in IBM Systems journal, 1966, 5, (2), pp. 78-101.

[20] UMASS Trace Repository. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[21] SNIA IOTTA Repository. [Online]. Available: http://iotta.snia.org/tracetypes/3