Efficient SSD Cache for Cloud Block Storage via Leveraging Block Reuse Distances

Ke Zhou, *Member, IEEE*, Yu Zhang, Ping Huang, Hua Wang, Yongguang Ji, Bin Cheng, and Ying Liu

Abstract—Solid State Drives (SSDs) are popularly used for caching in large scale cloud storage systems nowadays. Traditionally, most cache algorithms make replacement upon each miss when cache space is full. However, we observe that in a typical Cloud Block Storage (CBS) system, 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 article, 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. LEA mainly employs two lists, Lazy Eviction List (LEL) and Block Identity List (BIL), which keep track of two types of victim blocks respectively based on their cache duration when replacements occur, to improve cache efficiency. When a cache miss happens, if the victim 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 greatly reduces the number of SSD writes.

Index Terms—Cloud block storage, cache algorithm, SSD, reuse distance

1 Introduction

Modern cloud storage systems are facing the challenge of concurrently serving thousands of tenants to server a wide variety of workloads [1], [2], which imposes stringent requirements on the performance. Cloud block storage (CBS), an emerging generic block-level storage service, has been widely deployed by major cloud providers due to its high concurrency, scalability, and reliability. However, due to structural reasons, there are a large number of random IO requests at the server-end of CBS, which greatly ruins the performance of the data server nodes which are mainly backed up by Hard Disk Drives (HDDs).

Solid State Drives (SSDs), commonly used as a caching layer, have been widely deployed in cloud block storages because of their superior random read and write performance. However, designing the most appropriate caching algorithms largely determines the performance of the storage systems since the miss penalty is very expensive and the latency of a missed request may be one or two orders of magnitude higher than that of a hit request.

Existing advanced cache algorithms make replacement on each miss when the cache space is full, which is mainly based on the principle of locality and designed for the host cache or client-end cache of cloud stroage systems. However, we find that at the server-end of a typical Cloud Block Storage [3], there is a great percentage of blocks having large reuse distances, which implies a lot of blocks will be re-referenced only in the distant future. In such a scenario, if a simple and naive replacement is made upon each miss, newly accessed blocks will pollute the cache without bringing any benefit to the hit ratio, reducing cache efficiency. In addition, we have also observed a similar phenomenon in other scenarios using public IO traces collected from other data centers. Unfortunately, as we will discuss later, existing cache algorithms lead to suboptimal cache efficiency when large reuse distances exist.

In this paper, to address the problem of large reuse distances in a cache system, we propose a novel cache algorithm, LEA, specially tailored for CBS and other similar scenarios. In contrast to conventional wisdom, LEA does not make a replacement by default when there is a cache miss unless certain (lazy) conditions are met. The conditions are defined by accounting in two aspects, i.e., the frequency of the newly accessed blocks and the value of the candidate evicted-block in the cache, where the “value” is defined as the importance or usefulness of this block for the cache (e.g., the candidate eviction block hasn’t resided in the cache for too long compared to its own reuse distance and has a high probability of generating a hit in the future). In doing this, the cache duration of blocks can be largely extended. More importantly, the number of writes to SSD can be largely reduced, prolonging the life time of the SSD.

The contributions of this study are as follows:

1) We propose a novel cache algorithm LEA which is suitable for cloud block storages and other similar

- *K. Zhou, Y. Zhang, P. Huang, and H. Wang are with the Wuhan National Laboratory for Optoelectronics and School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei 430074, China. E-mail: raidkick@263.net, {yuzhang2016, hwang}@hust.edu.cn, templestorager@temple.edu.*
- *Y. Ji, B. Cheng, and Y. Liu are with Tencent Corporation, Shenzhen, Guangdong 518054, China.*
  E-mail: {raidmanji, bencheng, burtliu}@tencent.com.

Manuscript received 13 June 2019; revised 29 Feb. 2020; accepted 26 Apr. 2020. Date of publication 11 May 2020; date of current version 26 May 2020.
(Corresponding author: Ping Huang.)
Recommended for acceptance by C. Ding.
Digital Object Identifier no. 10.1109/TPDS.2020.2994075

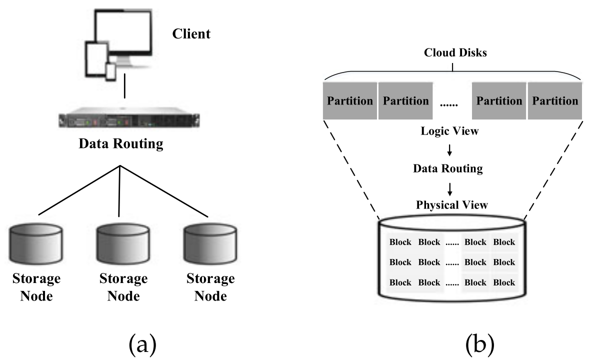


Fig. 1. The architecture (a) and data layout (b) of cloud block storage systems.

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.

2) The implementation of LEA has a low complexity O (1) due the employment of two simple lists (i.e., Lazy Eviction List and Block Identity List).

3) We have experimentally demonstrated the efficiency of LEA using IO traces collected from Tencent CBS as well as several public IO traces.

4) We have also used a high-temporal-locality-fidelity (HTF) sampling method to reduce the amount of the original trace data and demonstrated its effectiveness in maintaining the original workload characteristics (esp. the temporal locality characteristic) of sampling data.

The structure of this paper is as follows. In Section 2, we briefly introduce the background and motivation of this work. In Section 3, we give a description of our LEA algorithm. In Section 4, we present the IO collection method, the detail of the sampling method, as well as the experimental results via using an in-house developed simulator. In Section 5, we discuss the related work. Finally, in Section 6, we conclude the paper.

2 Background and Motivation

2.1 Cloud Block Storage

Cloud block storage provides tenants with a generic block-level storage service. Tenants can access this service via buying virtual machines and cloud disks supplied by cloud providers. Through this service, tenants can create partitions and make file systems just like operating local disks. CBS has the advantage of higher scalability, universality, as well as reliability compared with other storage systems. Nowadays, such block-level storage service, with broad prospects, have been supplied by most of the mainstream public cloud providers (e.g., AWS, Tencent and Alibaba).

Generally, the logical architecture of cloud block storage consists of three layers, namely, the client layer, the routing layer, and the storage layer (as shown in Fig. 1a). First, the client layer serves thousands of tenants who use a variety of devices (e.g., desktops, laptops, cell phones and so on) to issue data requests. Then, the routing layer is responsible for data mapping and forwarding, which provides access services to the storage layer on behalf of the client layer. Finally, the storage layer is the backend part of CBS and is made up of a great number of storage nodes.

Fig. 1b shows the data layout in a CBS. A partition, also known as a logical container, comprises multiple blocks. A partition is the basic unit of routing and failure recovery. Each partition belongs to only one cloud disk. A block is the basic unit of space allocation and recycling. By using this data organization, the unified logical address space seen by tenants can comprise blocks distributed on different storage nodes, and the routing information can be reduced considerably (benefiting from the concept of the partition), with the efficiency of data forwarding guaranteed as well.

More specifically, CBS physically consists of *client hypervisor*, *storage master* (or controller), and *storage server* (as shown in Fig. 2a). These three parts are connected by IP or FC network [4], [5]. The client hypervisor acts as the cloud disk controller. It has two functions. First, the client hypervisor is responsible for cloud disk virtualization. A cloud disk is composed of multiple blocks distributed on different storage nodes. The client is responsible for mapping physical blocks to a unified logical address. Second, the client hypervisor is used for storage protocol conversion to split IO requests and route the requests to different storage nodes. The storage master is responsible for managing the routing information of clients to data servers. To ensure data availability, the storage master also assumes the roles of server node management and replication management. It is responsible for monitoring the status of the data server. When a node has a bad disk or a

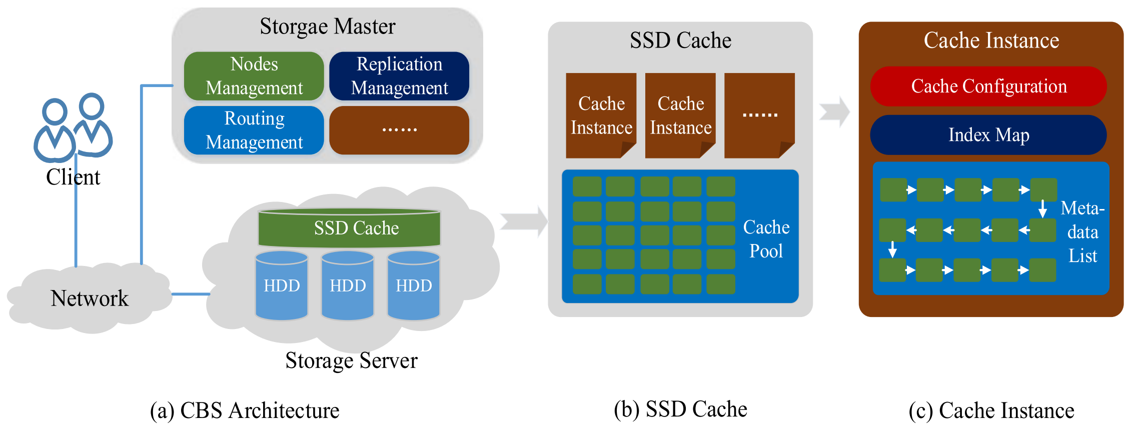


Fig. 2. The architectural overview of a cloud block storage with an SSD caching layer sitting atop of backend HDD storage system.

crash happens, the master should start data migration and perform recovery in a timely manner. The storage server is mainly responsible for data management and block allocation, and it is deployed on the storage nodes. Since the tenants' data blocks are distributed on different storage nodes in units of partitions, the requests at the data server are mostly random read and write requests, which are quite different from the host IO requests. This will greatly diminish the IO performance of the data server which is mainly constructed by the magnetic storage medium.

Adding an SSD cache (as shown in Fig. 2a) at the data server-end can effectively improve the random read and write performance. As shown in Fig. 2b, SSD cache server consists of several cache instances, sharing the SSD cache pool. Each cache instance (shown in Fig. 2c) consists of index map, metadata list, and other cache information. One instance serves for one or more storage nodes.

Upon receiving a read request from the client, the storage server first finds the data in the index map of the SSD cache. If found in the map, the data will be returned to the client directly. Otherwise, the data residing on the back-end HDD should be returned. A write-request is always first written to the SSD cache, and then flushed to the back-end HDD asynchronously. Obviously, the hit ratio of each cache instance largely affects the cache performance. In this paper, we introduce a caching algorithm, LEA, which greatly improves the performance of data server-end cache in CBS.

2.2 Reuse Distance in Cloud Cache

In a cache system, data reuse distance has a great impact on the cache hit ratio. Generally speaking, 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 data block is defined as the amount of unique data between two consecutive references to the same block [6]. It also has been called as inter-reference recency (IRR) [7]. For example, assuming a block sequence of 1-2-4-5-4-3-3-2, the reuse distance of Block 2 is the size of three blocks, because the unique data set between two consecutive references to Block 2 is {3,4,5}. There are also many studies (e.g., [8] and [9]) in which the reuse distance is defined as the amount of data between two consecutive references to the same block. According to this definition, the reuse distance of Block 2 is the size of five blocks, as the blocks referenced between the two consecutive accesses to Block 2 are Block 4, 5, 4, 3, and 3. In this study, we adopt the second definition of reuse distance for the sake of simplicity.

Traditional cache algorithms like LRU make replacement upon each miss when cache space is full, which is mainly guided by the principle of temporal locality. The main goal of these cache replacement algorithms is to retain blocks with smaller reuse distances and evict those with larger reuse distances. These algorithms can work efficiently when most of the data reuse distance is smaller than the cache size.

Unfortunately, based on our analysis of the cache traces, we find that the block reuse distance increases significantly in the context of cloud systems such as CBS. As shown in Fig. 3, the solid line represents the complementary cumulative distribution function of block reuse distance of the Tencent CBS accesses, while the dotted line with crosses, and with dots, are the distribution functions for host and memory IOs, respectively. We can get a better understanding of the result by looking at point A as marked in Fig. 3. Point A (32 GB, 50 percent) means that for CBS accesses, there are 50 percent of the blocks whose reuse distances are larger than 32 GB (similarly, point B means 50 percent distances are larger than 32 MB, and point C means 33 percent distances are larger than 64 KB). Therefore, if we use a cache algorithm like LRU, 50 percent of the reused blocks would not hit when the cache size is equal to or less than 32 GB. Equally said, it may require hundreds of gigabytes of cache size to cause the vast majority of reused blocks to be hit, which is too expensive.

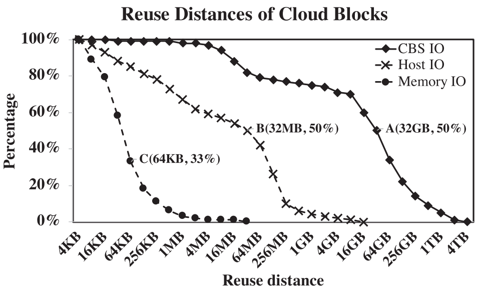


Fig. 3. Reuse distances of cloud blocks. This picture shows the complementary cumulative distribution of reuse distances.

This phenomenon is mainly attributed to 3 reasons:

1) Most data blocks with small reuse distances have been already cached and filtered by the client end cache devices.
2) The capacity of a single cloud disk can reach dozens of TB, which is several orders of magnitude higher than traditional disks. As a result, the cloud applications with larger data scale may induce larger reuse distances.
3) In a multi-tenant sharing cloud environment, the accesses from different tenants are intermixed, causing increased reuse distances.

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 a considerable number of blocks with small reuse distances, which are not captured by algorithms like MRU. Note that access patterns in CBS still fit in with the locality principle proposed by Peter Denning [10] and the hotspot working sets still exist in CBS workloads. However, the reuse distances of hotspots may be increased in our scenario.

2.3 A Look at Existing Cache Algorithms

Cache systems have been ubiquitously deployed in almost all computing systems to improve system performance. Researchers and pioneer engineers have developed various cache algorithms which are respectively suitable for their system environments, ranging from traditional on-chip cache to cloud cache. However, it could be sub-optimal to directly

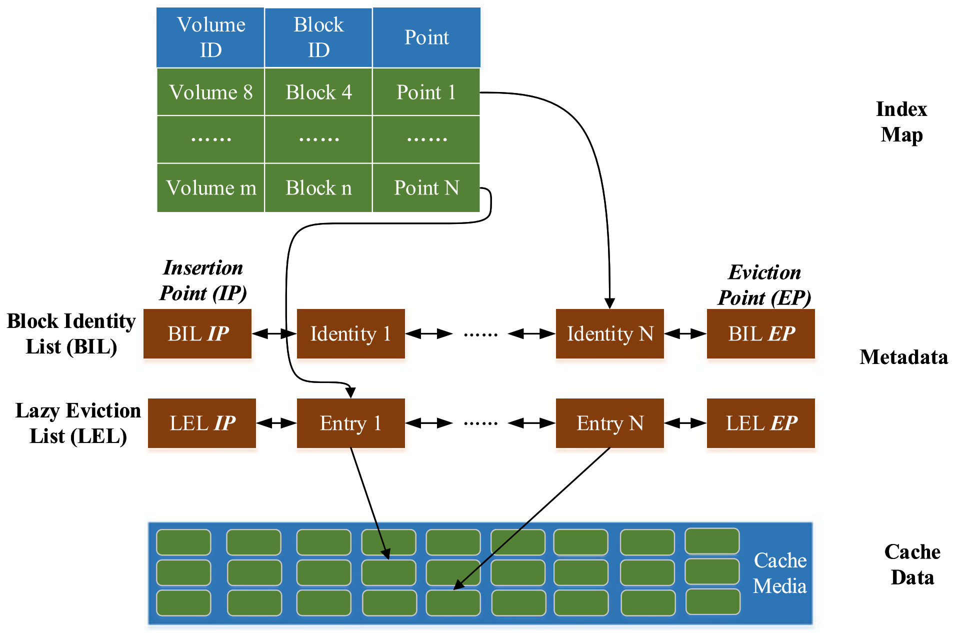


Fig. 4. The structure of LEA, comprises three layers, including index map, metadata, and cache blocks. The metadata layer employs two lists named Block Identify List (BIL) and Lazy Eviction List (LEL), which work in tandem to conduct cache replacement policy.

utilize existing cache algorithms to manage SSD cache for CBS due to the following two reasons.

First, most existing cache algorithms (e.g., LRU, MRU, LFU, CLOCK, FIFO and so on) make replacement upon 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 are re-referenced only after a long time, as revealed in Fig. 3. This observation is quite different from the reuse distance behaviors of Host and Memory level IO requests, which show much shorter reuse distances. 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 [11], EELRU [12], 2Q [13], MQ [5], ARC [14], LIRS [7] and FRD [6] 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 still suffer from a similar problem. By contrast, our proposed LEA algorithm does not make replacement upon each and every cache miss by default, which can largely extend the residence time of cached blocks, alleviating the problem.

Second, an SSD has limited write endurance. The number of writes to an SSD is an important factor when using SSDs, which should also be accounted for by SSD cache algorithms [15] [16] [17] [18]. There are many algorithms proposed to reduce the number of writes to the SSD and extend the lifetime of SSD in the cache context, such as LARC [8], S-RAC [9], and “One-Time-Access-Exclusion” policy [19]. They typically use a filter list to prevent blocks from entering the cache unless when blocks have been accessed for twice [8] or more times [9], or satisfied other conditions [19]. In doing this, they can eliminate a lot of useless write traffic to SSD. However, these algorithms always sacrifice hit ratio at the same time. LEA differs from these algorithms in the following three aspects:

1) Algorithms like LARC never allow a block to enter into cache upon its first access, while LEA may allow.
2) If a block has been re-referenced for twice or more times, it must enter the cache and replacement occurs with algorithms like LARC, but not in LEA.
3) When most of the block reuse distances are larger than the size of filter list, algorithms like LARC work poorly, whereas LEA algorithm is exempt from this problem.

Studies like [20] are also designed to reduce the number of writes to SSDs, but they are not applicable 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 is typically not available at CBS.

Studies like [19], [21], [22] have proposed a machine learning based method, which can accurately recognize and filter one-time-accessed files, to improve the efficiency of the photo cache service for a social network. However, to train the machine learning modes and ensure high accuracy, these methods typically rely on existing historical data, which is not available at the block level storage.

3 Design and Implementation

In this section, we introduce the design and implementation of our proposed LEA algorithm. At a high level, LEA does not make replacement by default when there is a miss unless when certain conditions are met. The conditions are defined by two factors, i.e., the accessed frequency of the missed blocks and the importance or usefulness of the candidate evicted-block in the cache. This algorithm is suitable for CBS and other similar scenarios where block reuse distance increases significantly.

As shown in Fig. 4, each cache instance consists of three parts, namely index map, metadata list, and cache data. The index map (e.g., using a hash table) is the data structure deployed for accelerating data queries. The metadata list records the metadata of cached blocks and is dynamically adjusted by cache algorithm. Next, we will describe the implementation of the cache algorithm in detail.

3.1 LEA Algorithm

The main effective working mechanism of LEA is enabled by using two lists, *Lazy Eviction List (LEL)* and *Block Identity List (BIL)*, both of which are of equal list length. As shown in Fig. 4, 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 record identities of blocks.

Algorithm 1. The Pseudo-Code of Lazy Conditions

```
1:  LAZY_1( )
2:  if LEL.EP.flag > 0 then
3:      return TRUE;
4:  end
5:  else
6:      return FALSE;
7:  end
8:      LAZY_2( )
9:  if LEL.EP.flag > 0 and (LEL.EP.age < LEL.EP.reuse_distance
    × LEL.EP.flag × K) then
10:     return TRUE;
11: end
12: else
13:     return FALSE;
14: end
```

An identity only includes the volume number and address information (offset) of a block. An entry contains extra block information such as *last_access*, *reuse_distance*, *age*, *flag*, etc. in addition to the same address information as that in BIL list. The information in an entry 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 its *reuse_distance*,[2] it indicates the block has a higher value 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 halves when the block is regarded as a block eviction candidate but has not yet been evicted due to the lazy replacement. If the *flag* is greater than 0, it indicates the block is valuable for the cache.

When a cache miss happens, LEA does not make replacement and only inserts the missed block identity into the BIL list if it meets the lazy conditions. Otherwise, it will perform replacement and insert the missed block entry into LEL list. The lazy conditions are defined by two aspects, i.e., the accessed frequency of the missed blocks and the "value" of the candidate evicted-block in the cache. More precisely, if a block has been hit in the BIL list, it has a higher probability of entering the LEL list. By doing this, we indirectly take accessed frequency into consideration. The "value" is defined as the importance or usefulness of this block for the cache and evaluated by using the block information (i.e., *last_access*, *reuse_distance*, *age*, *flag*, etc.) mentioned before. Here, we treat read and write misses the same.

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

2. Here, the reuse distance is the time between its last two accesses (and not, e.g., the average time between accesses).

There are two lazy conditions respectively used for the scenarios when it is the first access to the current referenced block or it is a re-access. Algorithm 1 shows the pseudo-code of the two lazy conditions. These two conditions are used to test whether LEL.EP should stay in the cache. It should be easy to notice 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 to be noted. First, the variable *LEL.EP.age* is the difference between variable *cnt* and *LEL.EP.last_access* and it means how long LEL.EP has been in the cache. Second, there is another lazy factor K in LAZY_2(), which also indicates the degree of laziness. A larger K means being lazier.

Algorithm 2 shows the pseudo-code of LEA algorithm. There are two lazy functions, i.e., LAZY_1() and LAZY_2(). These two lazy functions are used to judge whether it meets the lazy conditions or not.

Algorithm 2. The Pseudo-Code of LEA

```
1:  INITIALIZATION: Set cnt = 0
2:  Assume access to block X
3:  cnt++
4:  if LEL Miss then
5:      if BIL Miss then
6:          if LAZY_1( ) is TRUE then
7:              LEL.EP.flag > > 1;                      ▷ CASE 1
8:              move LEL.EP to LEL.IP;
9:              Evict BIL.EP;
10:             Insert X.identity into BIL.IP;
11:         end
12:         else
13:             Evict LEL.EP;                           ▷ CASE 2
14:             X.flag = PARA;
15:             X.last_access = cnt;
16:             Insert X into LEL.IP;
17:         end
18:     end
19:     else
20:         LEL.EP.age = cnt - LEL.EP.last_access;
21:         if LAZY_2( ) is TRUE then
22:             LEL.EP.flag > > 1;                      ▷ CASE 3
23:             move LEL.EP to LEL.IP;
24:             move X.identity to BIL.IP;
25:         end
26:         else
27:             X.flag = PARA;                          ▷ CASE 4
28:             X.last_access = cnt;
29:             move LEL.EP.identity to BIL.IP;
30:             Evict LEL.EP;
31:             move X to LEL.IP;
32:         end
33:     end
34: end
35: else
36:     X.flag++;                                       ▷ CASE 5
37:     X.reuse_distance = cnt - X.last_access;
38: end
```

When the cache space is not full, all the missed-blocks are inserted into LEL to fill the cache list. When the cache space is full, assuming that the currently accessed block is X, then one of the following five cases will happen.

- *CASE 1 :* X is not in LEL $\bigcup$ BIL but LEL.EP satisfies the lazy condition.

  This is a miss situation. We do not perform replacement because it meets the lazy conditions. We first halve the value of variable *LEL.EP.flag* and move the block at LEL.EP to the insertion point of LEL. Then we evict the identity at BIL.EP and insert the identity of X into the insertion point of BIL. We have to move the block from LEL.EP to LEL.IP because block at LEL.EP is the candidate evicted-block which may be evicted in a short time. We move this block from LEL.EP to LEL.IP to extend its cache duration.
- *CASE 2 :* X is not in LEL $\bigcup$ BIL and LEL.EP does not satisfy the lazy conditions.

  This is also a miss situation, but in this case we perform a replacement. We first evict LEL.EP. Then we set the value of variable *X.flag* as the lazy parameter PARA. Then we increase variable *cnt* by 1, and set the value of variable *X.last_access* as *cnt*. Finally, we insert X into the insertion point of LEL.

  We should point out two points. First, the variable *cnt* and *X.last_access* are used to calculate the reuse distance of X (CASE5). Second, the lazy parameter PARA is used to indicate the degree of laziness. Larger PARA means being lazier.
- *CASE 3 :* X is in BIL (but not in LEL) and it satisfies the lazy conditions.

  This is a miss situation. We do not perform replacement because it meets the lazy conditions. We first halve the value of variable *LEL.EP.flag* and move LEL.EP to the insertion point of LEL. Then we move the identity of X to the insertion point of BIL.
- *CASE 4 :* X is in BIL (but not in LEL) but it does not satisfy the lazy conditions.

  This is a miss situation. We perform a replacement because it does not meet the lazy conditions. We first move the identity of LEL.EP to the insertion point of BIL and evict LEL.EP. Then we set the value of variable *X.flag* as the lazy parameter PARA. Then we increase variable *cnt* by 1, and set the value of variable *X.last_access* as *cnt*. Finally, we insert X into the insertion point of LEL.
- *CASE 5 :* X is in LEL (but not in BIL).

  This is a hit situation. We only increase variable *X.flag* by 1 and update the value of variable *X.reuse_distance* as the difference between the variable *cnt* and *X.last_access*.

3.2 An Analysis of Why LEA Performs Well

Let us assume that $P_{in}(i)$ is the probability of a missed block being admitted to the cache and $H(i)$ is the number of hits of this block if it is inserted into the cache list. N is the number of total missed data blocks. Then hit ratio can be denoted as Eq. (1), and the number of writes to SSD cache can be presented as Eq. (2).

$$\begin{aligned} Hit\ ratio &= \frac{The\ number\ of\ hits}{The\ number\ of\ total\ access} \\ &= \frac{\sum_{i=1}^{N} P_{in}(i) \times H(i)}{The\ number\ of\ total\ access} \end{aligned} \tag{1}$$

$$The\ number\ of\ writes \propto \sum_{i=1}^{N} P_{in}(i). \tag{2}$$

From the above equation, we can see there is a positive correlation between hit ratio and $\sum_{i=1}^{N} P_{in}(i) \times H(i)$. The number of writes to SSD cache is in proportion to $\sum_{i=1}^{N} P_{in}(i)$. $P_{in}(i)$ is less than or equal to 1.

For traditional cache algorithms, such as LRU, MRU, and so on, all the missed data blocks are inserted into the cache list. Therefore, $P_{in}(i)$ is equal to 1 and the number of writes to SSD cache is at the maximum value.

Different from those algorithms, LEA does not make a replacement by default when there is a cache miss unless certain (lazy) conditions are met. Therefore, $P_{in}(i)$ is less than 1, thus decreasing the number of writes. However, this does not necessarily mean the hit rate will be decreased. This is because the hit ratio is also related to $H(i)$. From the discussion in Section 2, we understand that there is a great percentage of blocks with large reuse distances in our scenario. LEA can extend the cache duration of blocks, thus increasing $H(i)$. Therefore, in our scenario, we have the opportunity to improve the hit ratio while ensuring smaller number of writes, which has been demonstrated by our later experiment. To improve the controllability of LEA, we use the parameters $PARA$ and K to adjust the values of $P_{in}(i)$ and $H(i)$. Therefore, different $PARA$ and K values lead to different performance.

Some previous SSD-friendly cache algorithms like LARC can also filter some unnecessary write traffic to SSD cache by decreasing $P_{in}(i)$. However, these algorithms do not reduce $H(i)$ and thus they do not perform well in terms of hit ratio in later experimental results.

4 Experimental Results

We have evaluated the performance of LEA algorithm using an in-house trace-driven simulator BlockCacheSim [23], which we have made publicly available. This simulator is implemented based on a public block level flash cache [24]. In this section, we present the simulation results.

We have compared LEA with other four algorithms, including LRU, ARC, LARC as well as OPT [25]. LRU is a traditional algorithm. ARC is a state-of-the-art algorithm, which can accurately identify the useful blocks for cache by using two self-tuning lists. LARC is an SSD-friendly algorithm, which can reduce write traffic to SSD considerably. OPT is an optimal algorithm which unfortunately cannot be realized in practice. However, OPT provides the theoretical upper limit hit ratio for a fixed-sized cache. Overall, LEA achieves the highest hit ratio in most situations (shown in Table 1). More importantly, LEA can also significantly reduce the number of writes to the cache devices (shown in Table 2). Tables 1 and 2 are the average hit ratio and SSD write number of all experimental results, and more detailed results are shown in Section 4.3.

TABLE 1
The Average Hit Ratio of All Experimental Results

	LRU	LARC	ARC	LEA	OPT
tencent_0	0.394	0.426	0.498	0.542	0.619
tencent_1	0.449	0.467	0.524	0.581	0.662
tencent_2	0.161	0.172	0.201	0.273	0.312
tencent_3	0.209	0.160	0.221	0.295	0.428
tencent_4	0.409	0.424	0.445	0.476	-
websearch	0.423	0.471	0.492	0.513	0.685
adsds	0.632	0.641	0.648	0.657	0.843
online	0.480	0.589	0.563	0.615	0.683
stg	0.566	0.547	0.581	0.613	0.671

The hit ratio result of the OPT algorithm for tencent_4 is omitted due to its excessive computational complexity.

TABLE 2
The Average SSD Write Number of All Experimental Results

	LRU	ARC	LEA	LARC
tencent_0	2.316	2.299	1.302	1.000
tencent_1	2.089	2.082	1.270	1.000
tencent_2	5.331	5.321	1.739	1.000
tencent_3	4.791	4.805	1.938	1.000
tencent_4	2.261	2.256	1.138	1.000
websearch	4.988	4.377	1.879	1.000
adsds	2.253	2.156	1.460	1.000
online	2.253	2.156	1.460	1.000
stg	2.253	2.156	1.460	1.000

The number of writes of other algorithms is normalized to that of LARC algorithm.

4.1 Traces

4.1.1 Trace Collection

To demonstrate the efficiency of LEA, we have collected IO traces from Tencent CBS using a proxy server, while ensuring not to affect tenants' perceived performance. The proxy server is positioned near the storage nodes where the cache is located. Furthermore, LEA is also applicable to other scenarios with relatively large block-reuse-distance. To demonstrate this, we also tested LEA on public IO traces collected from other data centers [26][27]. The characteristics of the IO traces are given in Table 3.

We characterize these IO traces from four aspects, i.e., total traffic, unique data, read ratio and re-access ratio. Among them, re-access ratio is the rate of re-accessed traffic to the total traffic. This metric has a great impact on hit ratio.

As shown in Table 3, *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 a one week-long trace which is tested to evaluate the consistency of LEA algorithm.

Other traces are from open source repositories. *websearch* is from a web search engine [26]. *adsds* is a trace collected over a period of 24 hours for display ads platform data server [27]. *online* is from a course management system of FIU [27] and *stg_0* is for web staging [27].

4.1.2 Sampling Method

The amount of original data collected from Tencent CBS is specially large, reaching the scale of petabyte magnitude. So we use a high-temporal-locality-fidelity (HTF) sampling method to reduce the amount of test data. This sampling method is somewhat similar to SHARDS [28], [29]. The detail of HTF is shown in Table 4.

To demonstrate the efficiency of HTF, we have compared it with other three sampling methods, M1 [30], M2 [31] and M3 (also shown in Table 4). Referencing to related works on IO workloads identification [32], [33], [34], [35], [36], [37], we have evaluated the difference between the sampling methods and the original data from the following six aspects :

1) Read Ratio (IO). It is the read ratio in IO count. We measure the sampling efficiency of this dimension by calculating the read ratio difference between the sampling traces and original traces. Smaller difference indicates better.
2) Read Ratio (Throughput). It is the read ratio in term of throughput.
3) IO Size. It is the distribution of IO sizes. We calculate the euclidean Distance of IO size distributions to represent sampling efficiency. Smaller distance means better.
4) Re-Access Ratio. Re-access ratio, which has a great impact on hit ratio, is the rate of re-accessed traffic to the total traffic.
5) Temporal Locality. We use reuse distance distribution to represent temporal locality of each IO workload. The sampling efficiency of this dimension is also measured by the euclidean Distance. Smaller distance means better.

TABLE 3
Characteristics of IO Traces

Trace	Total Traffic	Unique Data	Read	Re-Access	Trace
Name	(GB)	(GB)	Ratio	Ratio	Description
tencent_0	149.91	33.26	13.95%	77.81%	write-intensive
tencent_1	156.67	32.07	8.05%	79.53%	write-intensive
tencent_2	108.48	63.90	68.44%	41.10%	read-intensive
tencent_3	82.20	40.50	42.49%	50.73%	read-write balanced
tencent_4	9972.14	1033.29	19.72%	89.64%	1 week trace from CBS
websearch	65.82	14.37	99.99%	78.17%	web search
adsds	48.47	3.73	96.84%	92.30%	display ads platform data server
online	54.53	2.09	21.80%	96.15%	course management system of FIU
stg	23.26	6.37	31.76%	72.66%	web staging

TABLE 4
Sampling Methods

Method	Description	Efficiency
HTF	This method maps some attributes (e.g., volume id, request offset and so on) in every IO record to a value calculated using a specific hash function. The IO record will be selected only when the first several digits of this value are equal to specified values.	HTF performs well (especially in keeping the original temporal locality characteristic) when the sampling ratio is greater than or equal to 1 percent.
M1	This method generates a random number range from 0 to 1 with a uniform distribution. The sampling is determined by the range of this random number.	By using various sampling ratios, this method performs well in terms of keeping the original read ratio and IO size characteristics.
M2	This method samples the records with a fixed IO interval.	It is similar to M1.
M3	This method samples the records with a fixed throughput interval.	This method has a poor performance even when the sampling ratio is very large.

TABLE 5
Differences Between Original Data and Sampling Data

Sampling Ratio	Sampling Method	Reading Ratio (IO)	Reading Ratio (Throughput)	IO Size	Re-Access Ratio	Temporal Locality	Spatial Locality
2%	HTF	2.19%	0.48%	0.01	15.01%	0.18	47.67%
	M1	0.05%	0.08%	0.00	33.14%	0.44	59.24%
	M2	0.04%	0.09%	0.00	33.22%	0.44	64.07%
	M3	0.04%	0.09%	0.68	33.22%	0.33	64.07%
1%	HTF	2.05%	0.28%	0.01	16.23%	0.19	45.80%
	M1	0.03%	0.08%	0.00	40.23%	0.49	66.61%
	M2	0.11%	0.06%	0.00	40.30%	0.50	69.59%
	M3	1.03%	44.66%	0.69	27.80%	0.37	169.11%
0.2%	HTF	29.76%	60.14%	0.05	16.50%	0.15	19.54%
	M1	0.16%	0.17%	0.00	57.18%	0.57	79.85%
	M2	0.10%	0.04%	0.00	57.31%	0.57	80.26%
	M3	1.64%	45.29%	0.69	39.37%	0.46	195.66%
0.1%	HTF	56.33%	129.73%	0.12	14.32%	0.09	10.75%
	M1	0.20%	0.40%	0.00	65.03%	0.60	81.37%
	M2	0.03%	0.43%	0.00	65.03%	0.60	83.35%
	M3	3.64%	46.44%	0.70	46.25%	0.51	212.99%
0.01%	HTF	17.25%	23.61%	0.02	18.15%	0.29	11.83%
	M1	0.02%	2.74%	0.00	85.87%	0.68	86.99%
	M2	0.66%	3.40%	0.00	85.80%	0.68	87.80%
	M3	5.96%	47.76%	0.71	66.44%	0.62	267.53%

6) Spatial Locality. We use average seek distance, which is the difference between request offsets of two consecutive IO, to indicate spatial locality.

Table 5 gives the comparison results. Overall, we can notice that HTF performs pretty well in most cases. Moreover, in our following experiments, we use the 1 percent sampling ratio for HTL as it represents trade-off point delivering very good results on most of the metrics.

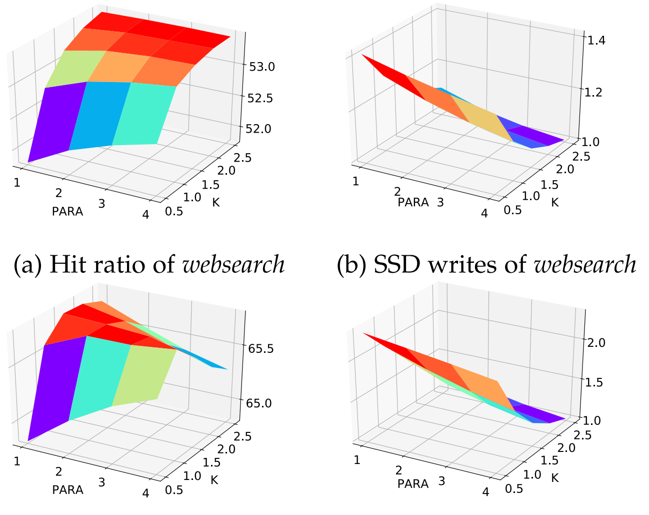
In this work, a 1 percent sampling ratio can reduce the amount of data to an acceptable level. As can be seen from Table 5, when the sampling rate is equal to or greater than 1 percent, HTF sampling method performs well in terms of reading ratio, IO size distribution as well as temporal and spatial locality. In this work, we use the HTF method with a 1 percent sampling ratio, which is reasonable.

4.2 The Sensitivity of Parameters

As mentioned in Section 3.1, there are two lazy parameters PARA and K in the LEA algorithm. Generally, a larger PARA or K means being lazier, and the number of writes to SSD will be reduced (as can be seen from Figs. 5b and 5d). However, using larger lazy parameters do not necessarily mean LEA would work more efficiently because it may influence the hit ratio. Figs. 9a and 9c, considering hit ratio, the optimal values of PARA and K are respectively 2 and 2.5 for trace *websearch*. Whereas, the best settings of PARA and K are 2 and 1 respectively for trace *adsds*. From our comprehensive experiments, we find that for most workloads, the appropriate values of PARA and K are 2 and 1, which results in a higher hit ratio and reduces the SSD writes as well for most of our target workloads. Therefore, in all our experiments, we set PARA and K as 2 and 1, respectively.

4.3 Evaluation

In this section, we give a detailed description of the experimental results. Fig. 6 shows the results of hit ratio, while Fig. 7 shows the results of the number of writes. It should



(a) Hit ratio of *websearch* (b) SSD writes of *websearch*

(c) Hit ratio of *adsds* (d) SSD writes of *adsds*

Fig. 5. The sensitivity of parameters by using trace *websearch* and *adsds*. Figure (a) and (c) are the results of the hit ratio. Figure (b) and (d) are the result of SSD writes. The numbers of writes are normalized to the minimum value of all values.

be noted that the numbers in Fig. 7 are normalized to the result of the LARC algorithm.

Figs. 6a and 7a show the results for *tencent_0*. The cache size is set from 0.8 GB to 4 GB.[3] As can be seen from Fig. 6a, LEA constantly achieves the highest hit ratio compared to other algorithms. On average, the hit ratio of LEA (54.23 percent) exceeds that of LRU (39.40 percent) by 37.64 percent, LARC (42.61 percent) by 27.27 percent and ARC (49.79 percent) by 8.92 percent. When it comes to the number of writes to SSD, LEA is 43.07 percent lower than LRU and 42.63 percent lower than ARC. LARC has the lowest write traffic, which is 23.66 percent lower than LEA. However, 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. As for the results of *tencent_1* (Figs. 6b and 7b), on average, the hit ratio of LEA (58.06 percent) exceeds that of LRU (44.86 percent) by 29.42 percent, LARC (46.70 percent) by 24.33 percent and ARC (52.39 percent) by 10.82 percent. As for the number of writes to SSD, LEA is 38.85 percent lower than LRU and 38.65 percent lower than ARC.

Figs. 6c and 7c show the results of *tencent_2*. The cache size ranges from 3.8 GB to 19 GB. On average, the hit ratio of LEA (27.29 percent) exceeds that of LRU (16.11 percent) by 69.40 percent, LARC (17.22 percent) by 58.48 percent and ARC (20.10 percent) by 35.77 percent. As for the number of writes to SSD, LEA is 66.70 percent lower than LRU and 66.45 percent lower than ARC. For the results of *tencent_3*, on average, the hit ratio of LEA (29.51 percent) exceeds LRU (20.93 percent) by 40.99 percent, LARC (15.98 percent) by 84.67 percent and ARC (22.05 percent) by 33.83 percent. As for the number of writes to SSD, LEA is 54.58 percent lower than LRU and 54.68 percent lower than ARC.

It gets the best results when using *tencent_2* as for the improvement of the hit ratio (shown in Fig. 6c). This is because for *tencent_2*, a great percentage of block reuse distances are larger than 32 GB (shown in Fig. 8 which illustrates the statistics of reuse distance for the workloads.). When the cache size is smaller than 32 GB and an LRU-like algorithm is used, most of the blocks cannot be cached or hit when re-referenced. By contrast, LEA well captures the characteristics of large block reuse distance, resulting in better results. Similar observations can be made in *tencent_3*.

Figs. 6e, 6f, 6g, and 6h and Figs. 7e, 7f, 7g, and 7h show the results of the open source IO traces *websearch*, *adsds*, *online* and *stg_0*. We can see from the results, LEA always delivers the highest hit ratio versus other algorithms, and LEA can also reduce the number of writes to SSD significantly compared to LRU and ARC. We should point out that LEA has a lower hit ratio than ARC when using trace *stg_0* and setting the value of cache size as 140MB (shown in Fig. 6h). This is because for trace *stg_0*, only a small part of block reuse distances are larger than 140MB. The cache size is big enough and the phenomenon mentioned before is weakened, so the promotion effect of LEA is limited. However, in practice, the cache size cannot be set big enough due to cost considerations. In the meanwhile, LEA can still reduce the number of writes to SSD in this situation (Fig. 7h).

4.4 Run-Time and Memory Overhead

The time complexity of LEA is O(1). From our experimental results, LEA costs 1.37 μs upon processing each data block on average. The run-time overhead of LEA is much lower than that of ARC (2.52 μs) and LARC (1.65 μs) and approaches that of LRU (1.35 μs). Please note that this overhead is evaluated by using our cache simulator which is implemented by C++ language. The processing time of each block reaching microsecond level is mainly because we have used the unordered_map in C++ STL to index the cache data blocks, and the searching and insertion operations of the index consume most of the time. Even so, the time consumption is still negligible compared to that of data access from HDD (on the order of milliseconds) or SSD (on the order of microseconds).

As for the memory overhead, the metadata of LEA cache entries contains four more data attributes (i.e., *last_access*, *reuse_distance*, *age*, and *flag*.) than referenced algorithms. In our implementation, each attribute is stored as a 4-bit nonnegative integer and the cache size is 128 GB (with 4 KB block size), and LEA will use 128 MB extra memory to store these attributes. Similarly, the BIL list of LEA will use 64 MB extra memory to distinguish the identities of different data blocks.

4.5 The Consistency of LEA

To test the consistency of LEA, we have used a week-long (from December 4th to December 10th, 2016) IO trace *tencent_4* collected from Tencent CBS. Figs. 9a and 9a show the results. The cache size is set as 2.4 GB. It should be noted that we used the sliding average to make the curve clearer. Specifically, we set the window length as 9 for the sliding average and the change to result is slight and negligible.

As can be seen from Fig. 9a, LEA always has the highest hit ratio compared to other algorithms. On average, the hit

3. This is the result of sampled IO traces, and the cache size should be set in a range from hundreds to thousands of GB in practice.

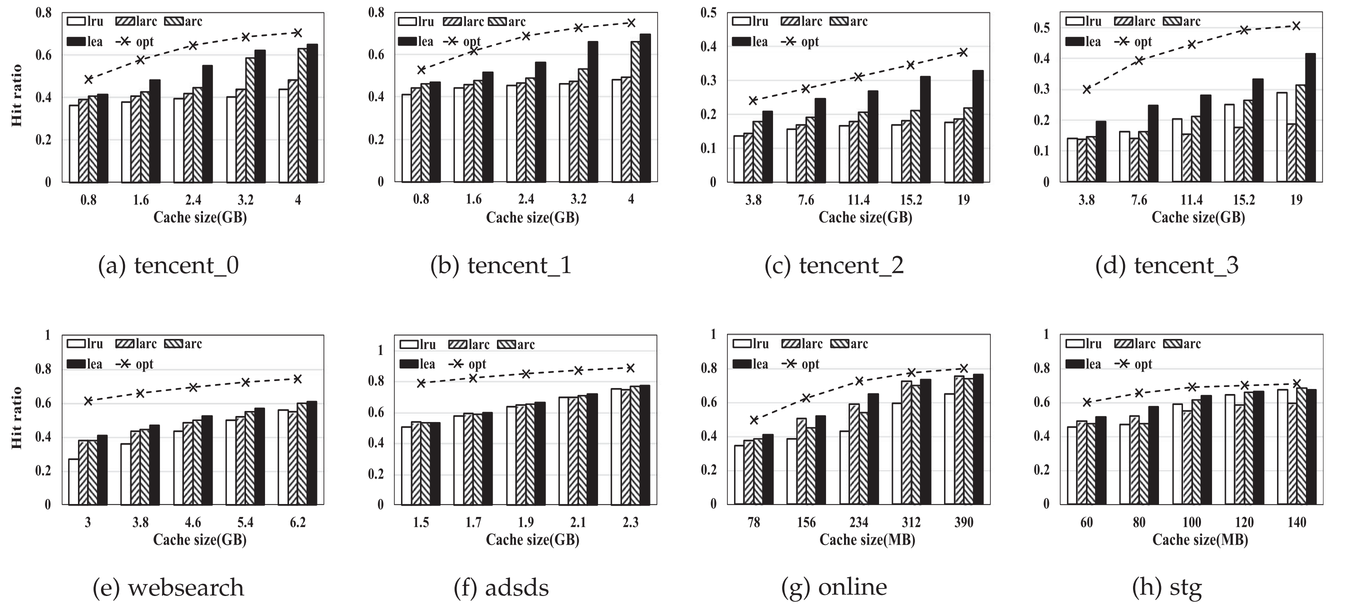


Fig. 6. The comparsion of hit ratio for different cache algorithms and workloads.

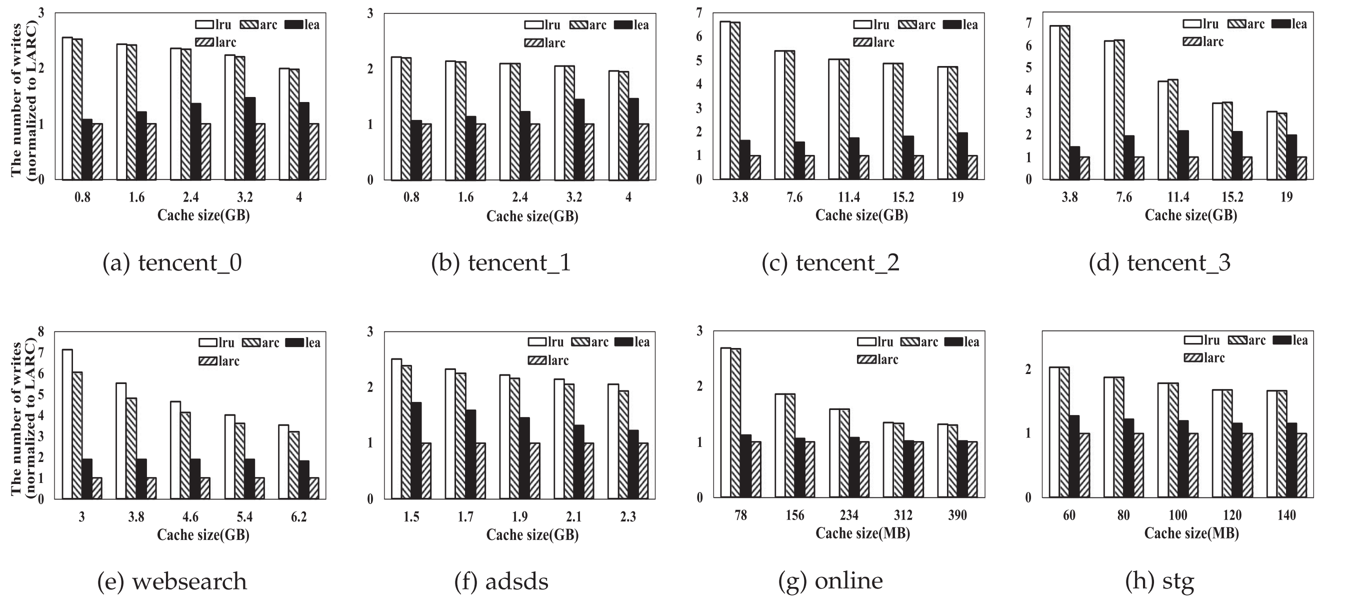


Fig. 7. The comparison of the number of writes to SSD for different cache algorithms and workloads.

ratio of LEA (47.63 percent) exceeds LRU (40.90 percent) by 16.45 percent, LARC (42.39 percent) by 12.32 percent and ARC (44.48 percent) by 7.08 percent.

There is also good performance for LEA as for the number of writes to SSD. On average, the number of writes to SSD using LEA is 50.64 percent lower than LRU and 50.21 percent lower than ARC. The number of writes using LARC is 12.69 percent lower than that of LEA. However, this gain comes at the expense of hit ratio.

We have also evaluated the average IO latency by using the simulator. In the simulator, we set the read latency of SSD t_{read_ssd} = 200 μs, read latency of HDD t_{read_hdd} = 14 ms, write latency of SSD t_{write_ssd} = 800 μs, and write latency of HDD t_{write_hdd} = 6 ms. t_{read_ssd}, t_{read_hdd}, t_{write_ssd} and t_{write_hdd} are the results from our experiments with fio [38] using 4 KB random IO. As shown in Fig. 9c, LEA always has the lowest latency. On average, the IO latency of LEA is 4.5 percent lower than ARC, 5.9 percent lower than LARC and 7.2 percent lower than LRU.

4.6 Extended Discussion

LEA can also be combined with previous SSD-friendly algorithms to further improve performance. For example, CFLRU [39] presents the Clean-First LRU algorithm which intentionally delays the eviction of dirty pages in DRAM memory replacement to reduce unnecessary write traffic (induced by dirty page flush operations) to SSD storage. The optimization goal of CFLRU does not conflict with our LEA algorithm which is implemented at SSD storage level. Therefore, aiding with LEA, it enhances the performance of

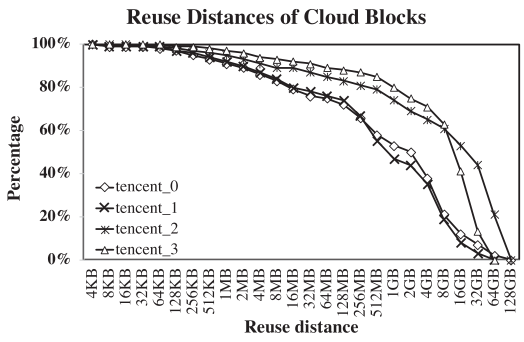
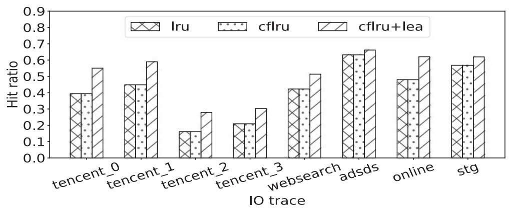


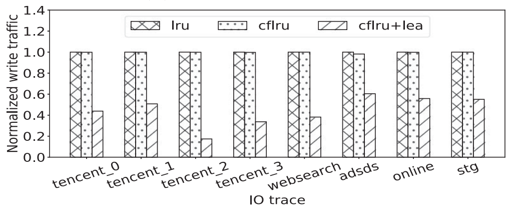
Fig. 8. Reuse distances of cloud blocks. This picture shows the complementary cumulative distribution of reuse distances for the workloads.

CFLRU at the two-level cache scenario where DRAM memory and SSD are deployed as the two-level cache for the HDD storage devices.

As shown in Fig. 10, we compared the hit ratio and the number of writes to SSD of LRU DRAM + LRU SSD (lru), CFLRU DRAM + LRU SSD (cflru), as well as CFLRU DRAM + LEA SSD (cflru+lea). The SSD cache size is set to be the same configuration as Section 4.3 and the DRAM cache size is set to be 1 percent of the SSD cache. All the results are averages of multiple tests with different cache sizes. In the experiment, CFLRU delivers negligible performance improvements to pure LRU. And the hit ratio of CFLRU combined with LEA (51.71 percent) exceeds CFLRU (41.45 percent) by 24.75 percent. As for the number of writes to SSD, CFLRU combined with LEA is 55.47 percent lower than CFLRU.



(a) hit ratio result



(b) the number of writes to SSD

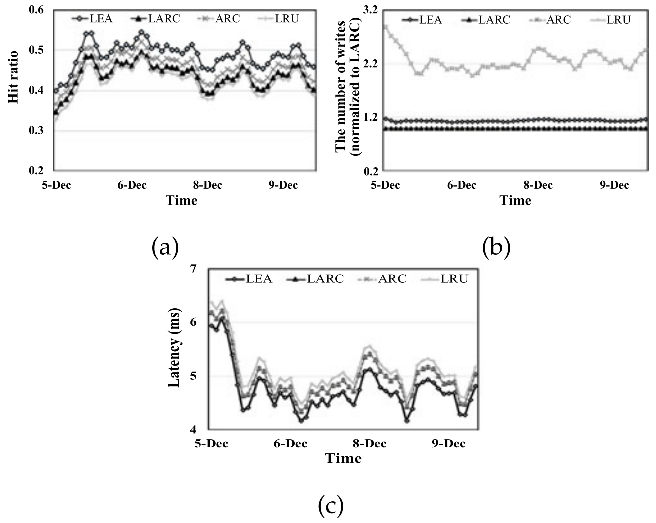
Fig. 10. The hit ratio (a) and the number of writes to SSD (b) results by combining CFLRU with LEA.

5 Related Work

Traditional cache algorithms, such as LRU, MRU, LFU, FIFO and so on, use requests' recency or frequency to identify valuable blocks for cache. LRU evicts blocks which are least recently used. However, when the reuse distances of most blocks are larger than the cache size (e.g., the loop pattern and CBS access pattern), this algorithm works poorly. Although algorithms like MRU are suitable for loop pattern, they don't work well for CBS access patterns. This is because there are still a lot of blocks with small reuse distance for CBS accesses, which cannot be taken into consideration by algorithms like MRU. Algorithms like LFU, LRFU [40] and FBR [41] are frequency based, which have the complexity $O(logN)$ to implement. Our LEA algorithm can work well for both loop pattern and CBS access pattern, and LEA uses two simple lists (i.e., Lazy Eviction List and Block Identity List), whose complexity is $O(1)$.

Traditional cache algorithms work very efficiently on chips, where the workloads typically exhibit good locality. However, cache is also used in some other scenarios like cloud storage whose IO workloads have a weaker locality. Some advanced algorithms are proposed for these scenarios. For example, LRU-K [11], SLRU [42], EELRU [12], 2Q [13], MQ [5], LIRS [7] and FRD [6] use multiple, fixed-size queues to distinguish valuable blocks with strong locality from blocks with weak locality. Algorithms like ARC [14] use multiple, self-tuning queues to automatically control the proportion of valuable blocks in cache according to different workloads. To adapt to workloads with weak locality, these algorithms (e.g., LIRS, ARC and FRD) always employ ghost lists to help track the history of IO sequences. However when the block reuse distance is larger than the size of ghost lists, these algorithms also won't work. LEA can largely extend the residence time of blocks in cache and thus improve cache performance.

LRU Insertion Policy (LIP) places an incoming data block in the LRU position. To overcome the inefficiency of LIP which is not sensitive to the changes in the working set, study [43] proposed Bimodal Insertion Policy (BIP). However, BIP is detrimental to cache performance for LRU-friendly workloads. Therefore, study [43] proposes a Dynamic Insertion Policy (DIP) to dynamically switch between LRU and BIP according to their cache hits. Not Recent Used (NRU) is analogous to LRU and has been widely deployed on modern high-performance processors. NRU adds an extra bit (nru-bit) for each cache block. When nru-bit is zero, it means this block is likely to be re-referenced soon, while, one indicates this block will not be re-referenced in the near future. By means of the extension of the NRU, RRIP [44] can quantitatively classify blocks with different re-reference intervals. That is, blocks with small RRPVs are likely to be re-referenced sooner than blocks with large RRPVs. For these algorithms,



(a) (b) (c)

Fig. 9. The hit ratio (a), the number of writes to SSD (b) and the latency results (c) by using *tencent_4.*

all the missed data blocks are inserted into the cache list, inducing a large number of unnecessary writes to SSD devices.

Furthermore, with the rapid development of SSD devices, many vendors use SSD as cache to improve the performance of storage systems (e.g., [45], [46], [47], [48], [49]). However, it is inappropriate to simply use conventional cache algorithms mentioned before to manage SSD cache. This is because SSD has two unpleasant features. One is that SSDs have limited P/E cycles and reducing write traffic to SSD can extend the devices' service life. The other is that SSD has asymmetric read-write performance and it has relatively poor random write performance [9]. So reducing the number of writes to SSD cache can improve overall performance of the system.

There are many algorithms proposed to reduce the number of writes to the SSD. Such as LARC [8] and S-RAC [9]. They both use a filter list to prevent useless data from entering the cache. In doing this, they can reduce a lot of useless write traffic to SSD. For example, LARC only allows hit blocks in the filter list to enter the cache. However, such algorithms also sacrifice some hit ratio. Moreover, when the reuse distance is larger than the size of the filter list, these algorithms also won't work. LEA can not only guarantee cache hit ratio, but also can reduce the number of writes significantly. Algorithms like RIPQ [50] reduce the GC overhead of SSD by aggregating data with similar locality. However RIPQ targets read-only workloads [9]. From the experimental results, we can conclude that LEA also works well in write-intensive workloads.

There are also some other methods (e.g., FlashTier [51], LAST [52] and SFS [53]) proposed to extend the lifetime of SSD. However these methods concentrate on optimizing in the flash translation layer (i.e., FTL).

6 Conclusion

SSD-based cache has 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 many times as possible. More importantly, when applied to SSD-based cache, LEA can significantly reduce the number of writes to the SSD devices, which results in an extended lifetime of the devices. LEA has a very low overhead of complexity of O(1). LEA has been tested by using an in-house trace-driven simulator with multiple IO traces collected from several data centers. Experimental results show that LEA can not only outperform the state-of-the-art cache replacement algorithms for hit ratio with a very low run time overhead but can also reduce the number of writes to SSD significantly.

Acknowledgments

This work was supported by the Innovation Group Project of the National Natural Science Foundation of China under Grant 61821003. The authors would also like to thank Tencent Technology (Shenzhen) Company Ltd. for experimental environment and IO trace supporting. The preliminary version of this article appeared in the proceedings of the 36th International Conference on Computer Design (ICCD), 2018.

References

[1] C. Z. Loboz, "Cloud resource usage: Extreme distributions invalidating traditional capacity planning models," in *Proc. 2nd Int. Workshop Sci. Cloud Comput.*, 2011, pp. 7–14.

[2] M. Stokely, A. Mehrabian, C. Albrecht, F. Labelle, and A. Merchant, "Projecting disk usage based on historical trends in a cloud environment," in *Proc. 3rd Workshop Sci. Cloud Comput.*, 2012, pp. 63–70.

[3] Tencent, "Tencent CBS," 2018. [Online]. Available: https://cloud.tencent.com/

[4] R. Hat, "Ceph," 2018. [Online]. Available: https://ceph.com/

[5] Y. Zhou, J. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proc. USENIX Annu. Tech. Conf.*, 2001, pp. 91–104.

[6] S. Park and C. Park, "FRD: A filtering based buffer cache algorithm that considers both frequency and reuse distance," in *Proc. 33rd IEEE Int. Conf. Massive Storage Syst. Technol.*, 2017, pp. 1–12.

[7] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proc. Int. Conf. Meas. Model. Comput. Syst.*, 2002, pp. 31–42.

[8] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, "Improving flash-based disk cache with lazy adaptive replacement," *ACM Trans. Storage*, vol. 12, no. 2, pp. 8:1–8:24, 2016.

[9] Y. Ni, J. Jiang, D. Jiang, X. Ma, J. Xiong, and Y. Wang, "S-RAC: SSD friendly caching for data center workloads," in *Proc. 9th ACM Int. Syst. Storage Conf.*, 2016, pp. 8:1–8:12.

[10] P. J. Denning, "The locality principle," in *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, Singapore: World Scientific, 2006, pp. 43–67.

[11] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "An optimality proof of the LRU-*K* page replacement algorithm," *J. ACM*, vol. 46, no. 1, pp. 92–112, 1999.

[12] Y. Smaragdakis, S. F. Kaplan, and P. R. Wilson, "The EELRU adaptive replacement algorithm," *Perform. Eval.*, vol. 53, no. 2, pp. 93–123, 2003.

[13] T. Johnson and D. E. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 439–450.

[14] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.

[15] P. Huang, P. Subedi, X. He, S. He, and K. Zhou, "FlexECC: Partially relaxing ECC of MLC SSD for better cache performance," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 489–500.

[16] P. Huang, G. Wu, X. He, and W. Xiao, "An aggressive worn-out flash block management scheme to alleviate SSD performance degradation," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 22:1–22:14.

[17] K. Zhou *et al.*, "Demystifying cache policies for photo stores at scale: A tencent case study," in *Proc. 32nd Int. Conf. Supercomputing*, 2018, pp. 284–294.

[18] K. Zhou, S. Hu, P. Huang, and Y. Zhao, "LX-SSD: Enhancing the lifespan of nand flash-based memory via recycling invalid pages," in *Proc. IEEE 33rd Int. Conf. Massive Storage Syst. Technol.*, 2017, pp. 1–13.

[19] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou, "Efficient SSD caching by avoiding unnecessary writes using machine learning," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 82:1–82:10.

[20] S. Kim, H. Kim, S. Kim, J. Lee, and J. Jeong, "Request-oriented durable write caching for application performance," in *Proc. USENIX Conf. Usenix Annu. Techn. Conf.*, 2015, pp. 193–206.

[21] F. Gelli, T. Uricchio, M. Bertini, A. Del Bimbo, and S.-F. Chang, "Image popularity prediction in social media using sentiment and context features," in *Proc. 23rd ACM Int. Conf. Multimedia*, 2015, pp. 907–910.

[22] A. Khosla, A. Das Sarma , and R. Hamid, "What makes an image popular?" in *Proc. 23rd Int. Conf. World Wide Web*, 2014, pp. 867–876.

[23] Y. Zhang, "Blockcachesim," 2018. [Online]. Available: https://github.com/zydirtyfish/BlockCacheSim

[24] M. Srinivasan, "Flashcache," 2018. [Online]. Available: https://github.com/facebookarchive/flashcache

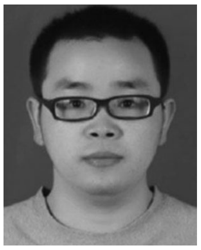
[25] L. A. Belady, "A study of replacement algorithms for virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.

[26] UMASS, "Umass trace repository," 2018. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[27] S. N. I. Association, "SNIA IOTTA repository," 2011. [Online]. Available: http://iotta.snia.org/tracetypes/3
[28] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, "Cache modeling and optimization using miniature simulations," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 487–498.
[29] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad, "Efficient MRC construction with SHARDS," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 95–110.
[30] D. Eklov and E. Hagersten, "Statstack: Efficient modeling of LRU caches," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, 2010, pp. 55–65.
[31] E. Berg and E. Hagersten, "Statcache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. IEEE Int. Symp.- ISPASS Perform. Anal. Syst. Softw.*, 2004, pp. 20–27.
[32] J. Layton, "IO pattern characterization of HPC applications," in *Proc. 23rd Int. Symp. High Perform. Comput. Syst. Appl.*, 2009, pp. 292–303.
[33] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application I/O signatures from noisy server-side traces," in *Proc. 12th USENIX Conf. File Storage Technol.*, 2014, pp. 213–228.
[34] J. Basak, K. Wadhwani, and K. Voruganti, "Storage workload identification," *ACM Trans. Storage*, vol. 12, no. 3, pp. 14:1–14:30, 2016.
[35] O. I. Pentakalos, D. A. Menascé, and Y. Yesha, "Automated clustering-based workload characterization," in *Proc. Joint 6th NASA Goddard Space Flight Center Conf. Mass Storage Syst. Technol. and Proc. 5th IEEE Symp. Mass Storage Syst.*, College Park, MD, 1996, pp. 1–11.
[36] O. Rodeh, H. Helman, and D. D. Chambliss, "Visualizing block IO workloads," *ACM Trans. Storage*, vol. 11, no. 2, pp. 6:1–6:18, 2015.
[37] S. Shen, V. van Beek, and A. Iosup, "Statistical characterization of business-critical workloads hosted in cloud datacenters," in *Proc. 15th IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2015, pp. 465–474.
[38] S. Media, "Fio tool," 2015. [Online]. Available: http://freshmeat.sourceforge.net/projects/fio
[39] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compilers Archit. Synthesis Embedded Syst.*, 2006, pp. 234–241.
[40] D. Lee *et al.*"LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies," *IEEE Trans. Comput.*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.
[41] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, 1990, pp. 134–142.
[42] R. Karedla, J. S. Love, and B. G. Wherry, "Caching strategies to improve disk system performance," *IEEE Computer*, vol. 27, no. 3, pp. 38–46, Mar. 1994.
[43] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," *ACM SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 381–391, 2007.
[44] A. Jaleel, K. B. Theobald, S. C. Steely Jr , and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 60–71, 2010.
[45] C. Albrecht *et al.*, "Janus: Optimal flash provisioning for cloud storage workloads," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 91–102.
[46] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "Cloudcache: On-demand flash cache management for cloud computing," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 355–369.
[47] Q. Huang, K. Birman, R. Van Renesse , W. Lloyd, S. Kumar, and H. C. Li, "An analysis of facebook photo caching," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 167–181.
[48] EMC, "VNX fast cache: A detailed review," 2013. [Online]. Available: https://www.emc.com/collateral/software/white-papers/h8046-clariion-cel erra-unified-fast-cache-wp.pdf
[49] Intel, "Intel® RAID: SSD cache with fastpath I/O," 2020. [Online]. Available: http://www.intel.com/content/www/us/en/servers/raid/raidssd-cache.html
[50] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, "RIPQ: Advanced photo caching on flash for facebook," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 373–386.
[51] M. Saxena, M. M. Swift, and Y. Zhang, "Flashtier: A lightweight, consistent and durable storage cache," in *Proc. 7th ACM Eur. Conf. Comput. Syst.*, 2012, pp. 267–280.
[52] S. Lee, D. Shin, Y. Kim, and J. Kim, "LAST: Locality-aware sector translation for NAND flash memory-based storage systems," *Operating Syst. Rev.*, vol. 42, no. 6, pp. 36–42, 2008.
[53] C. Min, K. Kim, H. Cho, S. Lee, and Y. I. Eom, "SFS: Random write considered harmful in solid state drives," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 12.

Ke Zhou (Member, IEEE) is a professor with the School of Computer Science and Technology, HUST. His main research interests include computer architecture, cloud storage, parallel I/O, and storage security. He has more than 50 publications in journals and international conferences, including the *IEEE Transactions on Parallel and Distributed Systems*, the *A Page Endurance Variance Aware*, SIGMOD, FAST, USENIX ATC, MSST, ACM MM, INFOCOM, SYSTOR, MASCOTS, ICC, etc. He is a member of the USENIX.

Yu Zhang is currently working toward the PhD degree with the Wuhan National Laboratory for Optoelectronics and School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China.



Ping Huang main research interests include non-volatile memory, operating system, distributed systems, DRAM, GPU, Key-value systems, etc. He has published papers in various international conferences and journals, including SYSTOR, NAS, MSST, USENIX ATC, FAST, DATE, IPDPS, Eurosys, IFIP Performance, INFOCOM, SRDS, MASCOTS, the *IEEE Transactions on Parallel and Distributed Systems*, the *ACM Transactions on Storage*, etc.

Hua Wang is an associate professor with the Wuhan National Laboratory for Optoelectronics, HUST. Her main research interests include computer architecture, cloud storage and intelligent storage. She has published papers in various international conferences and journals, including ICPP, ICS, ICCD, MSST, CIKM, SYSTOR, MASCOTS, NAS, the Journal of Systems and Software, etc.

Yongguang Ji received the graduate degree from the Huazhong University of Science and Technology, China. He is currently an expert engineer with Tencent Corporation. For the past decade, he has been dedicated to the establishment and optimization of Tencent Cloud Block Storage Systems as well as Distributed File Systems. His main research interest includes cloud block storage systems, distributed file systems, etc.

Bin Cheng received the graduate degree from the Huazhong University of Science and Technology, China. He is currently an expert engineer with Tencent Corporation. Since joining Tencent, he has focused on the independent research and development of Tencent Cloud Block Storage Systems, SQL and NoSQL products, as well as Tencent Distributed File Systems. His main research interest includes database, cloud block storage systems, distributed file systems, etc.

Ying Liu received the graduate degree in communication and information systems from the Civil Aviation University of China, in 2004. He is an expert engineer with Tencent Corporation. He has been devoted to Tencent's cloud computing infrastructure construction and cultivated excellent cloud technical capabilities. His main research interest includes cloud computing and storage, CVM, virtual networks, etc.