A Differentiated Caching Mechanism to Enable Primary Storage Deduplication in Clouds

Huijun Wu, Chen Wang, Yinjin Fu, *Member, IEEE*, Sherif Sakr, *Senior Member, IEEE*, Kai Lu, and Liming Zhu

Abstract—Existing primary deduplication techniques either use inline caching to exploit locality in primary workloads or use post-processing deduplication to avoid the negative impact on I/O performance. However, neither of them works well in the cloud servers running multiple services for the following two reasons: First, the temporal locality of duplicate data writes varies among primary storage workloads, which makes it challenging to efficiently allocate the inline cache space and achieve a good deduplication ratio. Second, the post-processing deduplication does not eliminate duplicate I/O operations that write to the same logical block address as it is performed after duplicate blocks have been written. A hybrid deduplication mechanism is promising to deal with these problems. Inline fingerprint caching is essential to achieving efficient hybrid deduplication. In this paper, we present a detailed analysis of the limitations of using existing caching algorithms in primary deduplication in the cloud. We reveal that existing caching algorithms either perform poorly or incur significant memory overhead in fingerprint cache management. To address this, we propose a novel fingerprint caching mechanism that estimates the temporal locality of duplicates in different data streams and prioritizes the cache allocation based on the estimation. We integrate the caching mechanism and build a hybrid deduplication system. Our experimental results show that the proposed mechanism provides significant improvement for both deduplication ratio and overhead reduction.

Index Terms—Data deduplication, cache management, ghost cache, primary storage, cloud services

1 Introduction

Data deduplication is a technique that splits data into small chunks, hashes these data chunks to compute collision-resistant fingerprints, and uses the fingerprints to identify and eliminate duplicate chunks in order to save storage space. Deduplication techniques have achieved great successes in backup storage systems [1]. Recent studies show that duplicate data widely exist in primary workloads [2], [3], [4]. In the cloud computing scenario, primary storage workloads of applications running on the same physical machine are also observed having a high duplicate ratio[5]. For a cloud datacentre, there are significant incentives to remove duplicates from its primary storage for cost-effectiveness and competitiveness.

Existing data deduplication methods for the primary storage have two main categories based on when the deduplication is performed: *inline* deduplication techniques [4], [5], [6] and *post-processing* deduplication techniques [2], [7], [8]. The former performs data deduplication on the write path of I/O requests to immediately identify and eliminate duplicate writes, while the latter remove duplicate data in the background to avoid the performance degradation on the write path. Nevertheless, challenges remain for both categories of methods.

For an inline deduplication method, the fingerprint lookup is a bottleneck because the size of a fingerprint table is often larger than the physical memory size. While a backup storage system may be able to tolerate the delay of the disk-based fingerprint lookup, a primary storage deduplication system has to rely on caching to satisfy the latency requirement. The state-of-the-art techniques for inline primary deduplication [4], [5], [6] exploit temporal locality of primary workloads by using in-memory fingerprint caches to perform deduplication. These deduplication mechanisms are called *non-exact deduplication* since they do not guarantee that all duplicate chunks are eliminated. However, primary workloads do not always exhibit strong temporal locality [9], [10]. When multiple applications running in virtual machines are sharing the same physical primary storage system, it is likely that there are data streams with weak temporal locality. These data streams may interfere with each other and further weakening the temporal locality of the overall data stream. Caching fingerprints from a data stream with weak locality wastes the valuable cache space and affects the duplicate detection performance of other data streams with good locality. Caching algorithms have a significant impact on inline deduplication performance. For backup deduplication systems, the stream interference problem is addressed through reordering data streams [11], which is unrealistic for primary storage.

- *H. Wu, S. Sakr, and L. Zhu are with The University of New South Wales, Sydney NSW 2052, Australia, and Data61, CSIRO, Eveleigh NSW 2015, Australia. E-mail: {huijunw, ssakr, limingz}@cse.unsw.edu.au.*
- *C. Wang is with Data61, CSIRO, Eveleigh NSW 2015, Australia. E-mail: chen.wang@data61.csiro.au.*
- *Y. Fu is with Army Engineering University, Nanjing 210008, China. E-mail: yinjinfu@gmail.com.*
- *K. Lu is with National University of Defense Technology, Changsha 410073, China. E-mail: kailu@nudt.edu.cn.*

Manuscript received 11 July 2017; revised 8 Dec. 2017; accepted 3 Jan. 2018. Date of publication 8 Jan. 2018; date of current version 11 May 2018. (Corresponding author: Kai Lu.)
Recommended for acceptance by L. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2790946

Another important concern for primary deduplication is the read performance. Compared to backup storage systems, the ratio of read operations are much higher in primary storage. The disk fragmentation caused by deduplication is likely to reduce the read performance since many sequential reads may be split into several random small reads. Existing approaches [4], [5] address this by only eliminating duplicate sequences whose sizes are longer than a fixed threshold. However, a fixed threshold may not be optimal for different I/O patterns.

For a post-processing only deduplication method [2], [7], [8], there are two main limitations: First, duplicate chunks have already been written to disks before being eliminated, hence the peak storage use is not reduced. Moreover, the post-processing deduplication has a negative impact on the lifetime of SSD based primary storage. Second, the resource contention between post-processing deduplication and foreground applications can be a problem when there are a lot of duplicates to be eliminated.

Fusing the inline and post-processing phase together is able to make each phase complement each other and address the limitation of applying each phase alone. It is particularly useful for deduplication in a virtualized system running multiple applications. The inline phase can eliminate duplicates with good temporal locality while leaving the rest of duplicates to the post-processing phase to achieve exact deduplication. In the two-phase process, the fingerprint cache plays an important role in the workload distribution. An efficient caching mechanism is able to identify a significant number of duplicates in the inline phase so that less data needs to be written to disks, which improves I/O performance and reduces the workload of the post-processing deduplication.

Existing inline deduplication solutions for primary storage mainly use the LRU (Least Recently Used) cache to exploit the temporal locality of duplicates [4], [5]. However, LRU is not able to well handle data streams with mixed levels of temporal locality. This results in the uncertain performance in achieving a good inline deduplication ratio at the block layer. Some recent "ghost cache" based algorithms, such as LIRS (Lower Inter-Reference Recency Set Replacement Policy) [12] and ARC (Adaptive Replacement Cache) [13] are designed to deal with the weak locality problem. However, their effectiveness in the fingerprint cache management is not well studied. Although some previous research [14] has applied these cache algorithms for I/O deduplication, cache algorithms like ARC only perform marginally better than LRU. In this paper, we evaluate the performance of different algorithms in the fingerprint cache management. We show that LRU and "ghost cache" based algorithms may perform poorly in primary storage deduplication systems. This is mainly due to the memory overhead, as we explain in Section 4.

To address this problem, we propose a novel cache replacement algorithm (TLE-LRU) in this paper. TLE-LRU estimates the temporal locality of different data streams and turns the memory overhead into the affordable computing overhead. We integrate the locality estimation based fingerprint caching mechanism into the primary deduplication design and build a hybrid deduplication mechanism named SLADE (Stream Locality Aware DEduplication).

In the inline deduplication phase, we differentiate the temporal locality of different data streams using a histogram estimation based method. The estimation method periodically assesses temporal locality of data streams. Based on the estimation, we prioritize fingerprint cache allocation to favor data streams with good temporal locality. The mechanism significantly improves cache efficiency in the inline deduplication phase and reduces the workload in the post-processing deduplication phase. The post-processing deduplication phase only deals with the relatively small amount of duplicate data blocks that are missed in the inline deduplication phase. Compared to systems that solely rely on the post-processing deduplication, a highly efficient inline deduplication process greatly reduces the storage capacity requirement and contention in system resources. Moreover, to reduce the disk fragmentation which may hurt the read performance for primary storage systems while maintaining a certain deduplication ratio, SLADE only eliminates duplicate sequences whose sizes are above certain thresholds. The thresholds are adjusted for different data streams dynamically.

Overall, this paper makes the following contributions:

1) We propose a hybrid deduplication mechanism that fuses an efficient inline deduplication process and a post-processing deduplication process together for primary storage systems shared by multiple applications.
2) We evaluate the performance of "ghost cache" based algorithms on the block-layer deduplication systems for primary storage and reveal their non-trivial memory overhead in fingerprint cache management.
3) We propose a locality estimation based cache replacement algorithm which significantly improves the fingerprint caching efficiency in primary storage deduplication systems. The estimation method is able to exploit stream-based locality for better cache space allocation.

We evaluate SLADE using traces generated from real-world applications. The result shows that SLADE outperforms the state-of-the-art inline and post-processing deduplication techniques. For example, SLADE improves the inline deduplication ratio by up to 39.70 percent compared with iDedup in our experiments. It also reduces up to 45.08 percent disk capacity requirement when compared with the post-processing deduplication mechanism in our evaluation.

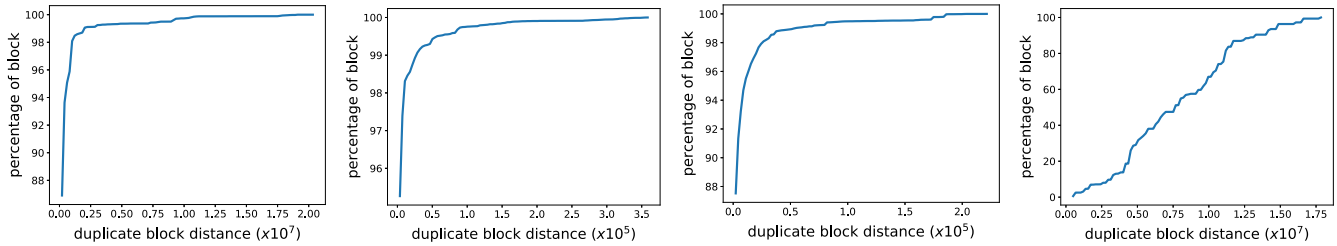
The remainder of this paper is organized as follows: Section 2 describes the background and motivations; Section 3 presents the design of SLADE; Section 4 analyses the limitations of using ghost cache in primary deduplication. Section 5 introduces our locality estimation method. Section 6 describes the read optimization method of SLADE. Section 7 presents the evaluation results; Section 8 reviews related work and Section 9 concludes the paper.

2 Background and Motivation

In this section, we present the background and key observations that motivate this work.

2.1 Deduplication for Primary Storage in Clouds

Virtualization techniques enable a cloud provider to accommodate applications from multiple users to run on a single physical machine while providing isolation between these



(a) FIU-mail trace (b) FIU-web trace (c) FIU-home trace (d) Cloud-FTP trace

Fig. 1. Temporal locality analysis for three I/O traces. CDF for the distance between two adjacent occurrences of the same-content data block. i.e., for a I/O sequence "abac", each letter represents a data block. The number of data blocks between two adjacent occurrence of "a" is 1. Note that the x-axis is scaled for each figure.

applications. Recently, container techniques like Docker [15] further reduce the overhead of application isolation, thus supporting running more applications simultaneously on a physical machine. These techniques raise many key challenges to the primary storage deduplication. In a typical configuration, the cloud software stack such as OpenStack [16] maps the data volumes of VMs to persistent block storage devices connected by networks. It is impractical to achieve deduplication within a container or a virtual machine due to the overhead of the storage device access and this approach is unable to identify duplicates across different VMs or containers. In addition, implementing the deduplication inside a VM still needs to involve the hypervisor in the virtual-physical block translation process. Hence, we place the deduplication functions in the host's hypervisor. A similar architecture is used in an existing post-processing primary deduplication system [17].

2.2 Temporal Locality and Fingerprint Cache Efficiency

Some recent studies reveal that locality of duplicates can be weak in primary storage workloads [10]. We evaluate temporal locality with real-world traces that contain a 24-hour I/O trace from a file server running in the cloud as well as the FIU trace [14] that is commonly used in the deduplication research. The file server is used for data sharing among a research group consisting of 20 people. We denote the file server trace as *Cloud-FTP*, the FIU mail server trace as *FIU-mail*, the FIU home trace as *FIU-home* and the FIU Web server trace as *FIU-web*.

TABLE 1
Workload Statistics of the 2-Hour Traces

Trace	Request number	Write request (%)	Duplicate writes (%)
FIU-home	293,605	91.03%	12.2%
FIU-mail	1,961,588	98.58%	84.4%
FIU-web	116,940	49.36%	52.8%
Cloud-FTP	2,293,424	84.15%	14.3%

TABLE 2
The percentage of Duplicates Detected in Each Stream Under Different Cache Replacement Algorithms

Cache Policy	FIU-home	FIU-mail	FIU-web	Cloud-FTP
LRU	4.58%	89.04%	3.71%	2.67%
LFU	4.67%	89.00%	3.75%	2.58%

As shown in Fig. 1, temporal locality of duplicates in primary storage systems varies among applications. The distances between two adjacent occurrences of a data block in all FIU traces are highly skewed and small in average, indicating good locality. However, temporal locality is weak in the *Cloud-FTP* trace. We further evaluate the cache efficiency when these workloads arrive at a storage system within the same time frame. LRU (Least Recently Used) and LFU (Least Frequently Used) are the cache algorithms used in existing primary deduplication systems [4], [5], [6]. Therefore, we use LRU and LFU in our evaluation.

We extract two-hour traces from FIU traces (10 am-12 am on the first day.) and Cloud-FTP trace. The characteristics of the two-hour traces are shown in Table 1. We mix these traces according to the timestamps of requests to simulate read/write requests from multiple applications in a cloud server. The cache size is set to 32K entries. Fig. 2 illustrates the actual percentage of cache space allocated to each data stream for LRU cache. LFU cache shows a similar pattern (not shown due to the page limit).

Table 2 shows the percentages of duplicate blocks detected in each stream under different cache policies. Specifically, for each stream, the percentage value indicates the ratio between the number of duplicates found in the stream and that found in all streams. Under both LRU and LFU, the cache allocated to Cloud-FTP stream is above 2/3 of the cache capacity (see Fig. 2), but the percentages of duplicates detected in the stream are only 2.67 and 2.58 percent, respectively. This shows that data streams with the weak temporal locality of duplicates result in poor cache efficiency and fail

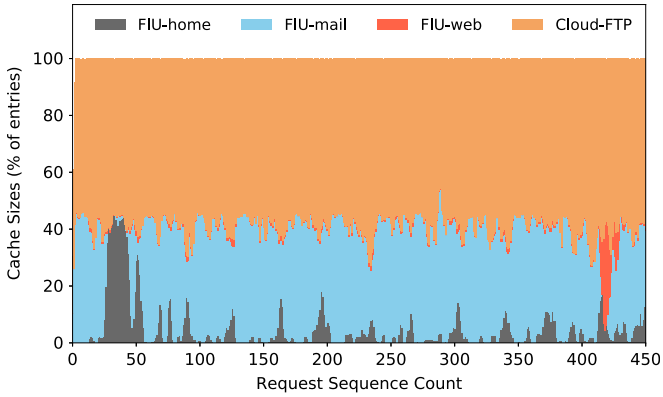


Fig. 2. The size of (LRU) cache entries occupied by each data stream. The x-axis is the request sequence count while the y-axis is the percentage of cache occupied by different streams. The area of different colors indicates the cache resources used by each data stream.

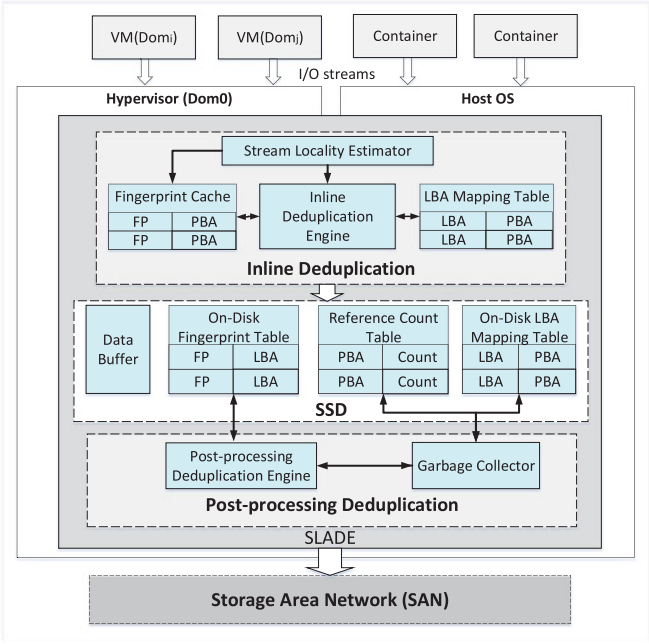


Fig. 3. System architecture of SLADE.

to detect most of the duplicates in inline deduplication. These duplicates are written to disks subsequently and require extra storage space when planning the capacity of the storage system. Improving cache efficiency is therefore important to reduce the storage capacity required in a deduplication system.

3 Hybrid Deduplication Architecture

Fig. 3 illustrates the system architecture of SLADE. Storage devices are connected to the server via networks like SAN. The hypervisor (e.g., Xen) is responsible for translating LBA (logical block address) to PBA (physical block address) for I/O requests from VMs to their host. The hybrid deduplication mechanism works at the hypervisor level to eliminate duplicate data blocks. For multiple containers running on the same host, the deduplication mechanism can be deployed at the block device layer of the host machine. For simplicity, we mainly use the hypervisor setting to describe the architecture for the rest of this paper.

3.1 Inline Deduplication Phase

In the inline deduplication phase, SLADE maintains an *in-memory fingerprint cache* that stores the fingerprint and PBA mapping to avoid the slow disk-based fingerprint table search, and an *LBA mapping table* that stores the mapping between LBAs and PBAs of blocks. The LBA mapping table can be stored in the battery-backed RAM to avoid data loss. The inline deduplication of data streams is performed in the *inline deduplication engine*: the fingerprint of each data block is computed by a cryptographic hash function, such as SHA-1. The deduplication engine then looks up the block fingerprint in the fingerprint cache. The *virtualization I/O stack* in the hypervisor is able to differentiate request streams coming from different VMs during the logical to physical translation. This enables the stream locality estimator to perform the per-stream estimation. The estimator keeps track of both the temporal and the spatial locality of data streams. The temporal locality estimation is used for optimizing the cache hit rate while the spatial locality estimation is used to adjust the deduplication thresholds for data streams in order to reduce the disk fragmentation.

When a fingerprint of the incoming data block has a cache hit, the LBA entry of the block and its corresponding PBA will be added into the LBA mapping table if such an entry does not exist, otherwise, the write of the block is treated as a duplicate. If the fingerprint has a cache miss, the data block is written to the underlying primary storage and the fingerprint may be inserted into the cache based on the caching algorithm. Note that the inline fingerprint cache does not prefetch fingerprints from the disk because the inline phase is latency-sensitive and the duplicates in primary workloads do not have good spatial locality. In this primary storage writing process, the data is staged in the SSD data buffer for performance consideration. We use the D-LRU [18] algorithm to manage the SSD data buffer and store recently accessed data in SSD to exploit temporal locality at the storage device level. The benefit of using the D-LRU algorithm is to prevent duplicate blocks from being written into the SSD cache. Using D-LRU can improve the performance of read operations and reduce the SSD device wear-out.

When a data block is written to the underlying primary storage, the corresponding metadata associated with this data block including its fingerprint, LBA, and PBA mapping as well as the reference count is updated in three tables in the SSD: on-disk fingerprint table, on-disk LBA mapping table, and reference count table. The duplicates whose fingerprints are not cached in the fingerprint cache are eliminated during the post-processing phase.

3.2 Post-Processing Deduplication Phase

In the post-processing deduplication phase, the deduplication engine scans the on-disk fingerprint table and identifies duplicates. The fingerprint table is implemented using a log-structured key-value store [19]. The $(fingerprint, LBA)$ pairs of data blocks are first appended to a log. The post-processing deduplication process runs in the background when the system is idle to insert fingerprints to key-value indices. During the insertion, the duplicate fingerprints are identified and the corresponding entries are removed. To alleviate the negative impact on read operations caused by the post-processing deduplication process, we maintain a priority queue to record the access order of data blocks and the deduplication process gives priority to old data blocks that have not been accessed for a while to avoid interfering with read requests.

When a duplicate entry is identified, the corresponding data block is removed from the storage and its LBA is mapped to the PBA of the identical block in the LBA mapping table. The reference count to the new PBA increases by 1 while the original PBA is removed. After the post-processing deduplication phase, the unique data blocks in the SSD data buffer are organized into fixed-sized coarse-grained objects and flushed to the underlying chunk store.

For storage devices like hard disk drives, data fragmentation has a significant impact on read performance. To balance this effect, we also only eliminate duplicate sequences whose sizes are larger than a threshold [4] in our current

implementation, in a way that is similar to the inline phase. The threshold is set by taking into account of the performance gap between sequential and random read performance for the underlying storage devices.

4 Ghost Cache in Inline Deduplication

The efficiency of the fingerprint cache is critical to a hybrid deduplication system, particular for workloads with mixed locality. Nevertheless, designing cache algorithms to deal with weak temporal locality is not a new problem. There are some work on optimizing cache replacement algorithms to deal with weak locality. Two representatives among them are LIRS (Lower Inter-Reference Recency Set Replacement Policy) [12] and ARC (Adaptive Replacement Cache) [13]. LIRS and ARC are both proposed to improve LRU. They both use extra memory, called *ghost cache* to extend the access history, thus better infering the locality of workloads. The cache replacement algorithms can then be adaptive to the change of temporal locality in workloads.

To our knowledge, there is no study investigating the effectiveness of LIRS and ARC-like algorithms on the fingerprint cache management in primary storage deduplication systems. In this section, we first give a brief description of LIRS (Section 4.1) and ARC (Section 4.2), and then analyze the limitations of applying them to the fingerprint cache management in block-layer primary storage deduplication systems (Section 4.3).

4.1 LIRS in Fingerprint Caching

LIRS uses a metric called IRR (Inter-Reference Recency) to make cache replacement decisions. IRR is the number of other distinct blocks accessed between two consecutive references to a block. When a block needs to be evicted, LIRS considers not only the recency of the block but also its IRR which reflects the access history of the block more accurately compared to recency.

LIRS uses ghost caches to remember the references to recently evicted data. All cache blocks are split into two parts. The blocks with high IRR are called HIR (High Inter-reference Recency) blocks while those with low IRR are LIR (Low Inter-reference Recency) blocks. LIR blocks occupy much more cache space compared to HIR blocks. The key idea of LIRS is to maintain two LRU lists to separately store the LIR and the HIR blocks. For a workload with weak duplicate temporal locality, the data blocks can only get into the HIR area and are evicted quickly if none of their duplicates arrive later. As a result, the majority cache space occupied by LIR blocks is not affected by data streams with weak temporal locality.

When a resident HIR data block is evicted from the cache, its ghost data block stays in the cache. The ghost block only stores the metadata (i.e., the block address in the page buffer management scenario) of the original block. If a subsequent data block is the same as the ghost data block, LIRS changes the status of the block from ghost to LIR again. The benefit of using a ghost cache is to keep a longer access history of data blocks. In comparison, LRU cannot memorize the occurrence of data blocks after they have been evicted from the cache. LIRS can, therefore, deal with data blocks with duplicate occurrences far away from each other, i.e., data streams with weak temporal locality. For the inline block-layer deduplication in primary storage, using LIRS may help to identify a duplicate fingerprint that is far away from its previous occurrence in the data stream.

4.2 ARC in Fingerprint Caching

ARC maintains two LRU lists T_1, T_2 to store the least-recently-used and least-frequently-used data items, respectively. The total size of T_1 or T_2 is C. Moreover, ARC maintains two extra ghost LRU lists B_1 and B_2 to record the references of the data items evicted by the LRU cache and LFU cache. The total size of the four LRU lists is $2C$. Similar to LIRS, the single-access data can only pass through T_1 and does not affect the frequently accessed data in T_2. The sizes of T_1 and T_2 are adaptive to the recency and frequency of the workloads. Specifically, when a data item in T_1 is accessed again, it would be moved to T_2. When data items in T_1 and T_2 are evicted, the data contents are also evicted. Nevertheless, the metadata (i.e., the addresses) items are kept in B_1 and B_2, respectively. If metadata items in B_1 and B_2 get hits later, the corresponding data items will be put back to the T_2 cache. Meanwhile, the number of hits in B_1 and B_2 are used to adjust the size of T_1 and T_2.

When applying ARC in a block-layer deduplication system, the fingerprint cache is able to dynamically adapt to the recency and the frequency change of workloads so that it can achieve a good balance between LRU and LFU. It is noteworthy that ARC still relies on at least moderate temporal locality of workloads. When temporal locality of a data stream is weak, neither the LRU or LFU list provides useful information to infer locality accurately.

4.3 Limitations of LIRS and ARC on Fingerprint Cache

LIRS and ARC both make use of the ghost cache to keep a longer history of data block accesses. For LIRS, the history helps cache replacement algorithm to deal with weak temporal locality. For ARC, the history can help the cache to achieve adaptive adjustment between LRU and LFU. However, the ghost cache approach has limitations when used for the fingerprint cache management in primary storage deduplication systems. This is mainly due to the memory overhead, as we explain below.

LIRS and ARC work well in many scenarios, especially in page buffer management of file systems where the data item in the cache is a page with a size of 4 KB by default. What are stored in the ghost cache are references to pages. To be specific, only a pair of $(inode \rightarrow i\_mapping, index)$ is stored for each page. Here, $inode \rightarrow i\_mapping$ is a pointer to an $address\_space$ structure while the $index$ is a $pgoff\_t$ type data indicating the desired position in the file. Therefore, to represent a 4 KB page in the ghost caches, only 16B reference is required. To limit the memory overhead, for LIRS, the number of entries in a ghost cache is often set to a factor (e.g., 100 percent) of the number of cache entries in the data cache. For ARC, the number of entries in a ghost cache is the same as that in the data cache. Overall, the memory overhead introduced by LIRS and ARC is small for page buffer management.

However, for the fingerprint cache in the block-layer deduplication, the data item in the cache is comparable in

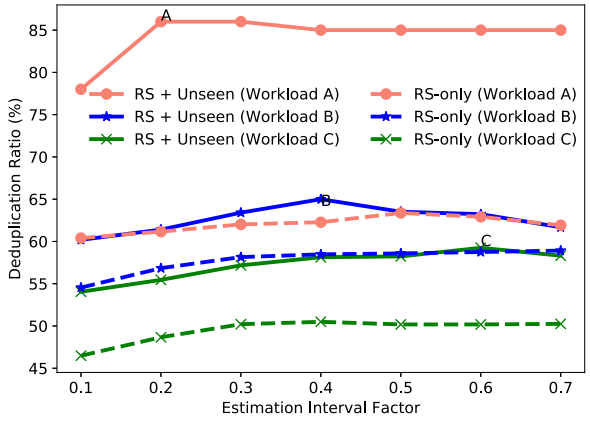


Fig. 4. Inline deduplication ratio versus estimation interval factor for three different workloads.

size with the data item in the ghost cache. The benefit of using LIRS and ARC is therefore affected by the relatively high memory overhead of maintaining the ghost cache. The fingerprint cache is organized as a map between the *fingerprint* and the *PBA*. A typical size of a *PBA* entry is 8 Bytes while the size of a fingerprint is 16 Bytes for MD5 or 20 Bytes for SHA-1. Therefore, when the references (fingerprints) of cached items are stored in the ghost cache after they are evicted, the memory overhead is non-trivial. The memory requirement for a data item in the ghost cache is almost the same as that in the actual data cache. This raises the question about whether the ghost cache approach is cost-effective for the fingerprint cache management.

On the other hand, the high memory overhead makes performance tuning of LIRS and ARC difficult. Specifically, when the workload already has good temporal locality for duplicates, the ghost cache has little impact on the cache efficiency but occupies memory that could be used for caching more fingerprints. While for the workload with weak locality, the ghost cache size should be configured larger accordingly. Without a systematic way to estimate the locality, it is easy to get a suboptimal deduplication ratio.

For the fingerprint cache management in a primary deduplication system, a feasible cache management scheme should be able to infer locality based on the information collected in a significantly less expensive manner. We achieve this with an estimation based method that incurs trivial memory overhead to differentiate locality of duplicates among data streams from different VMs or applications.

5 Temporal Locality Estimation

In this section, we give two caching algorithms for a fingerprint table: TLC-LRU (Temporal Locality Counting based LRU) is ghost cache-based and maintains counters for locality information; TLE-LRU (Temporal Locality Estimation based LRU) is based on the proposed locality estimation method for different data streams.

5.1 TLC-LRU Algorithm

The TLC-LRU algorithm uses the ghost cache to count the number of duplicates of data streams within certain time intervals to predict temporal locality of these streams.

Temporal locality of duplicates can be measured by the number of duplicates arriving in the storage system within a given time interval. The simplest method is to set an interval size n and count the number of occurrences of each fingerprint among the last n arrivals of a data stream. The fingerprints coming from each $Stream_i$ is assigned a separate cache space denoted by C_i. The size of C_i, denoted by $|C_i|$, is determined by the locality measure of the data stream. We define the number of duplicates in the interval as the *Temporal Locality Indicator* (TLI) which can be directly counted. The TLI value for a stream determines its priority. After obtaining TLI values for all streams, the following two strategies are applied. *First*, the fingerprints from data streams with a low TLI ($< CacheThreshold$) are not cached before the TLI of the stream increases (see Algorithm 1). *Second*, predicted TLI values for streams are maintained in a segment tree data structure called PrioriySegmentTree. The cached fingerprints belonging to streams with a lower TLI have a higher chance to be evicted. The evicting process is defined in Algorithm 2. The number of recent arrivals that have been taken into account, n, is normally smaller than the cache size due to duplicates. The memory overhead of storing the fingerprints depends on the number of duplicates among them. For streams with good temporal locality, the memory overhead is low as only the counts of duplicate fingerprints need to be stored.

Algorithm 1. TLC-LRU Pseudocode

Input: The fingerprints from streams.
f_{ij} is the jth fingerprint from stream i.
S_i is the segment representing stream i in the PrioritySegmentTree.
C_i is the cache space for stream i.

```
1  Procedure FingerprintProcess()
2    T_u = Threshold for updating PrioritySegmentTree;
3    Upon Event IntervalEnd && Σ(ΔS_i) > T_u
4      UpdatePrioritySegmentTree();
5    M = TotalCacheSize;
6    FingerprintCount(f_ij);
7    if len(S_i) > CacheThreshold then
8      if Σ_i|C_i| < M then
9        Insert f_ij into C_i;
10     else
11       f_pq, C_p = FingerprintToEvict();
12       Evict f_pq from C_p;
13       Insert f_ij into C_i;
14     end
15   end
```

However, the memory overhead can be high, particularly when overall locality of the aggregated stream is weak, which requires a larger interval size to make the fingerprint counts meaningful. We define *Estimation Interval Factor* as the ratio between interval size and the cache size. In our current implementation, the *Estimation Interval Factor* is adaptively set by monitoring the cost-efficiency between the achieved deduplication ratio and the memory cost. As shown in Fig. 4, the deduplication ratio changes with the *Estimation Interval Factor* for three workloads (see Section 7). A relative large *Estimation Interval Factor* is necessary to achieve high deduplication ratio. As a result, the high

memory overhead for handling streams with weak temporal locality compromises the overall deduplication ratio gain as the cache space for the fingerprints is squeezed.

Algorithm 2. TLC-LRU Subroutines

Input: Segment tree $T \rightarrow [S_1, S_2, \ldots, S_n]$ where the range size of S_i indicates the predicted TLI_i for S_i. n is the number of streams.

```
1 Subroutine FingerprintToEvict ()
2   Generate a random number r in total range of T;
3   foreach p in Range(1,n) do
4     if r is in S_p then
5       f_pq = EvictBySubCache(C_p);
6       return (f_pq, C_p);
7     end
8   end
```

5.2 TLE-LRU Algorithm

To address the memory overhead problem faced by the ghost cache based methods, we propose the TLE-LRU algorithm that trades the memory based counting for inferring locality with computing based locality estimation. In another word, TLE-LRU uses estimation rather than counting to obtain the number of duplicates in each interval n.

Specifically, TLE-LRU uses fingerprint samples from data streams in each interval to estimate TLI values of data streams. The theoretical basis of TLE-LRU is from an entropy estimation theory [20]. The theory shows that one can accurately estimate the entropy of a dataset by only using $O(\frac{n}{log_n})$ samples. We refer readers to [20] for the detail of the estimation theory. Algorithm 3 (TLI_Estimation) gives a brief description of how we estimate the TLI for a stream. Before introducing the algorithm, we define some concepts.

- *Reservoir* R_i is a vector that stores the fingerprint samples for $Stream_i$ in each interval.
- A *fingerprint frequency histogram (FFH)* for a stream in interval I is a histogram $f = f_1, f_2, \ldots$ where the value of f_i is the number of fingerprints that appear exactly i times in the interval.

Algorithm 3. TLI Estimation for $Stream_i$ in an Interval

Input: Reservoir R_i storing the samples of fingerprints for $Stream_i$.

```
1  Subroutine TLI_Estimation ()
2    Compute the transformation matrix T by binomial
     probabilities;
3    H_s = the observed FFH of samples;
4    H = the FFH of the whole fingerprints in the interval;
5    H'_s = T · H;
6    Solve the linear programming:
7      Objective function: min(Δ(H_s, H'_s)), in which
8        Δ(H_s, H'_s) = Σ_i 1/√(H_s[i]+1) |H_s[i] − (T · H)[i]|.
9      under constraints:
10       Σ_i H[i] = N
11       ∀i  H[i] > 0
12     return TLI_i = N_i − Σ_i H[i]
```

TLE-LRU uniformly samples fingerprints in each interval and stores the samples for $Stream_i$ in Reservoir R_i. This is consistent to the theoretical proof in [20]. The observed FFH for samples (H_s^o) of $Stream_i$ can be obtained from R_i. We define the FFH for all fingerprints of $Stream_i$ in the interval by H. The target of the estimation is to use H to compute the TLI_i. According to [20], the theoretical value of H_s (denoted by $H_s^{'}$) can be computed by $H_s^{'} = T \cdot H$ where matrix T is computed by a series of binomial probabilities. We minimize the distance between the observed H_s (H_s^o) and its theoretical value $H_s^{'}$ to estimate H. From H, we can then obtain the value of TLI_i.

It is noteworthy that the sampling method plays a critical role to achieve an accurate estimation. The estimation requires uniform sampling on the dataset being estimated. The original entropy estimation assumes a static dataset. However, the data comes as streams in inline deduplication. We address the problem and manage to achieve uniform sampling via Reservoir Sampling [21] which provides the guarantee that each fingerprint can be sampled with the same probability ($\frac{1}{k}$ where k is the sample size.) regardless of the interval size.

We use *TLI*s to guide the cache allocation for the corresponding streams. The fingerprints from a data stream with a higher predicted *TLI* are more likely to be kept in the cache than those from a data stream with a lower *TLI*. We use the historical *TLI* values accurately estimated by the *TLI* estimation algorithm to predict the *TLI*s of streams. A self-tuning double exponential smoothing method is used to achieve this prediction. Using the estimated $TLI(w-2)$, $TLI(w-1), \ldots$, we can predict $TLI(w)$ where w is the next *estimation interval*. The details of using the predicted *TLI* to guide the fingerprint cache management are as follows.

First, we propose a cache admission policy whereby fingerprints from streams with a low *TLI* are not cached if there exist streams with a much higher *TLI*. This strategy can avoid unnecessary caching of the fingerprints from data streams containing compressed data or other forms of compact data. For each individual stream, the cache can be managed by any caching policy.

Second, for fingerprints in the cache, we assign an evict priority value p_i to stream i, denoted by: $p_i = \frac{1}{TLI_i(w)}$. The evict priorities are mapped to adjacent non-overlapping segments in the segment tree. Specifically, stream i is represented by the segment $[\Sigma_{k=0}^{i-1} p_k, \Sigma_{k=0}^{i-1} p_k + p_i)$. When evicting a fingerprint, we generate a random number r and find the segment I to which r belongs. We then evict one cache entry in the space corresponding to the segment.

To show the effectiveness of TLI estimation, we compare its performance with a *reservoir sampling only* method, denoted by the RS-only, which directly estimates *TLI*s of data streams by the number of duplicate fingerprints sampled through the reservoir sampling. Fig. 4 gives the deduplication ratios of using RS-only or RS + *TLI* Estimation for *TLI* estimation under different estimation interval factors. It is clear that RS + *TLI* Estimation is able to achieve a much higher inline deduplication ratio with a smaller estimation interval compared with the RS-only method.

Moreover, a proper *estimation interval* value is important for achieving good cache efficiency. A large interval size may introduce outdated locality information. On the other hand, a small interval size cannot accurately capture temporal locality of a workload. We examine the impact of interval

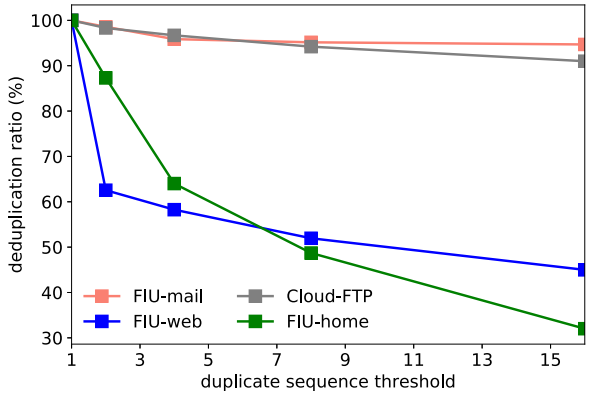


Fig. 5. Deduplication ratio versus threshold. Deduplication ratio versus threshold for different threshold of duplicate sequence length.

size in the following. We set the *estimation interval* to a factor of the number of fingerprint cache entries. The solid lines in Fig. 4 shows the inline deduplication ratio of SLADE for three different workloads (details are shown in Section 7) while choosing different *estimation interval factor*. The cache size is set to 160 MB. From workload A to C, the overall temporal locality for the workload decreases. The *estimation interval factor* needs to be set to a larger value for workloads with a worse temporal locality indicator value. One may note that the optimal *estimation interval factor*s for workload A, B and C are 0.2, 0.4 and 0.6, respectively. In practice, a good approximation is to set the *estimation interval factor* to $1 - d$ where d is the historical inline deduplication ratio for the aggregated stream.

The temporal locality estimation is triggered by the following events: 1. the finish of an *estimation interval*; 2. a significant drop of inline deduplication ratio; 3. the join or quit of virtual machines/applications.

6 Read Performance Optimization

To alleviate the disk fragmentation, some primary storage deduplication techniques only eliminate duplicate block sequences with length greater than a given *threshold* [4], [5]. However, for the applications with many small random I/Os, doing so may miss many duplicates [4]. In primary deduplication in the cloud scenario, the spatial locality of the workloads for different VMs may vary significantly. To explore the relationship between deduplication ratio and threshold, we analyze both the FIU traces and Cloud-FTP trace. As shown in Fig. 5, different workloads show different trends. When the threshold increases from 1 to 16, the inline deduplication ratio for FIU-mail and Cloud-FTP reduces by only 4.3 and 9.1 percent, respectively. The inline deduplication ratio for FIU-web drops by around 38.1 percent when the threshold increases from 1 to 2. When the threshold is 16, the inline deduplication ratio is 43.1 percent of that under a threshold of 1. For FIU-home trace, the inline deduplication ratio keeps dropping. When the threshold is set to 16, the inline deduplication ratio is only 32.0 percent of that under a threshold of 1.

Therefore, the threshold should be adaptive to different characteristics of data streams. It is noteworthy that there exists a tradeoff between the write latency and read latency while choosing a proper threshold. Write operations prefer a smaller threshold while read operations prefer a larger one. For writes, a smaller threshold indicates more comparisons before writing data to disks. For reads, a long threshold can avoid many random I/Os, thus reducing the read latency.

SLADE maintains two vectors V_w and V_r for each stream. V_w is used to store the histogram of the occurrence numbers of different lengths for duplicate write requests. V_r is used to store the histogram of different lengths for read requests. Both V_w and V_r have 64 items (bins). For instance, if $V_w[3] = 100$, it means that there are 100 duplicate write requests with a length of 3 blocks since V_w has been reset. If $V_r[3] = 100$, there are 100 read requests with a length of 3 blocks. Initially, the threshold is 16. The two vectors collect data when requests arrive. When the threshold update is triggered, given the histogram vector V_w and V_r, the threshold T is computed by $T = (1 - r) \cdot \overline{Len_d} + r \cdot \overline{Len_r}$ where $\overline{Len_d}$ and $\overline{Len_r}$ are the average lengths of duplicate write requests and read requests, respectively. T, therefore, is the balance point of the read and write latency. r is the ratio between the number of read requests and the number of all I/O requests. $\overline{Len_d}$ and $\overline{Len_r}$ are computed according to the information collected in V_w and V_r, respectively. To cope with the changes of the duplicate pattern for each stream, the two vectors are reset to all 0s when the total deduplication ratio decreases by over 50 percent since the last threshold update.

7 Evaluation

To evaluate the performance of SLADE and TLE-LRU, we use real-world traces to feed into SLADE. We compare SLADE with the following deduplication methods: the locality-based inline deduplication system (iDedup [4]), post-processing deduplication methods (e.g., [2], [17]) and the hybrid inline-offline deduplication system DIODE [22].

7.1 Configuration

The experiments are carried out on a server with a 16-core Intel E5-4660 CPU, 64 GB RAM, 256 GB SSD and 4 2 TB Seagate SATA 7200RPM HDDs organized as RAID-0 striping. The disks are connected through a SilverStone ECS01 Controller. One extra 2 TB HDD is used to store the unique block contents generated according to the hash values in the traces. We use the *FIU-home, FIU-web, FIU-mail* [14] traces in our evaluation. Moreover, we also collected a trace from a cloud FTP server (Cloud-FTP) used by our research group. The trace is obtained by mounting a network block device (NBD) as the working device for the file server, from which we capture read/write requests through the NBD-server.

To the best of our knowledge, there are no available larger scale I/O traces containing both fingerprints of data blocks and corresponding timestamps. We, therefore, use the four traces as templates to generate synthetic VM traces representing multiple VMs. Table 3 shows the statistics of the four workloads. The arrival order of requests in these workloads is sorted and merged based on timestamps. The generated trace has the same I/O pattern with the original traces. For the traces generated from the same template, the content overlap is randomly set to 0 - 40 percent which is the typical amount of data redundancies shared among users [23]. Each trace is replayed by a separate process

TABLE 3
Workload Statistics

Trace	# of block requests	Write request ratio	Duplicate ratio
FIU-home	2,020,127	90.44%	30.48%
FIU-mail	22,711,277	91.42%	90.98%
FIU-web	676,138	73.27%	54.98%
Cloud-FTP	21,974,156	83.94%	20.77%

according to the timestamps. Since we use one-day traces, it takes 24 hours to finish the replay.

We simulated a cloud host running 32 virtual machines. As shown in Fig. 1, FIU traces show better temporal locality compared with Cloud-FTP trace. We mix traces to form three workloads with different overall temporal locality. The ratios of data size between the good-locality (L) traces and weak-locality (NL) traces are around 3:1, 1:1 and 1:3 for workload A, B and C, respectively. The *estimation interval factor* is set to 0.5 at the beginning and is adjusted by the historical inline deduplication ratio dynamically for each workload.

7.2 Cache Efficiency in Inline Deduplication

We first compared the cache efficiency in inline deduplication between SLADE and iDedup. Specifically, we replayed the mixed workloads to simulate the scenario where multiple applications/services running on the same physical machine. Each I/O request for the three workloads is for a 4 KB block and MD5 is used as the hash function to calculate the fingerprints. Note that stronger hash functions like SHA-1 can be used and they may introduce higher memory overhead for sampling the fingerprints in TLE-LRU algorithm. Each entry of the deduplication metadata is 64 bytes and contains the fingerprint and the block address. According to the size and footprint of the traces, the total memory size for fingerprint cache is set from 20 MB to 320 MB in the experiments. Note that all the memory overhead is already counted into the cache size.

Here, we use inline deduplication ratio as the metric. It is defined as the percentage of duplicate data blocks that can be identified by inline caching. For the cache replacement policy of each stream, both TLC-LRU and TLE-LRU are used in the evaluation. Although iDedup [4] has claimed that LRU is the best cache replacement policy, we also implement ARC and LIRS in iDedup to understand the performance of ghost cache based fingerprint caches.

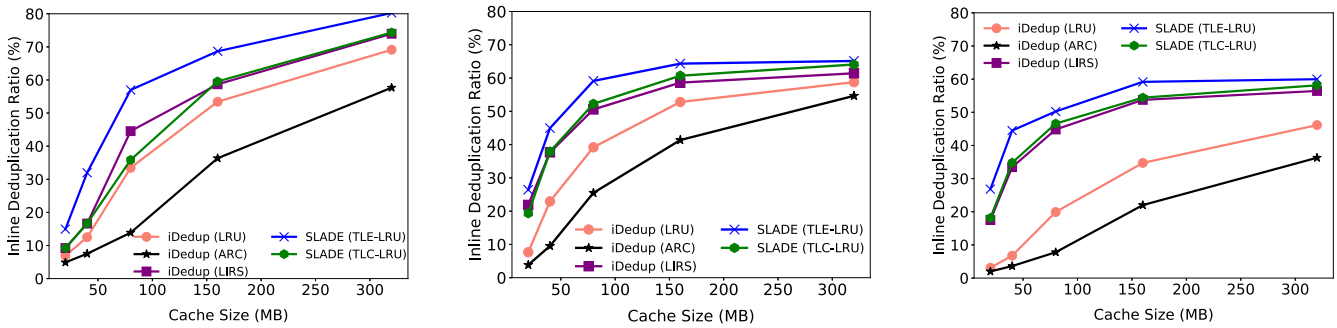
As shown in Fig. 6, ARC achieves the lowest inline deduplication ratio for the three workloads. ARC uses the ghost cache to find a balance between LRU and LFU. However, for the portion of workloads with weak temporal locality, this approach is ineffective with its low hit rates and memory use to infer locality. In the following experiments, SLADE refers to SLADE (TLE-LRU) unless otherwise mentioned. When the ratio of NL workloads increases, the inline deduplication ratio gap between SLADE and the original iDedup (iDedup with LRU) becomes larger. SLADE outperforms iDedup by 8.04 - 23.52, 8.09 - 30.19 and 13.86 - 37.75 percent for workload A, B and C, respectively. The reason is that TLE-LRU does not cache fingerprints when the temporal locality of a stream is weak. iDedup (LIRS) shows better performance compared to iDedup (ARC) and iDedup (LRU) because it is able to identify blocks which are unlikely to have a cache hit to avoid wasting cache space. TLC-LRU slightly outperforms LIRS due to its much lower memory overhead. For all the workloads, SLADE consistently outperforms iDedup under all cache configurations. Meanwhile, SLADE outperforms SLADE (TLC-LRU) by 7.95 - 62.20, 6.09 - 20.89 and 3.18 - 47.70 percent for workload A, B and C, respectively. The main reason is that TLE-LRU uses much lower memory to track the temporal locality for each data stream and manages the cache allocation according to the temporal locality estimation.

Overall, the weak locality in workloads results in a low inline deduplication ratio. With the locality estimation method in SLADE, the allocation of inline fingerprint cache dynamically gives the streams with better locality higher priority, thus improving the overall cache efficiency for inline deduplication.

7.3 Inter-Reference Distance Distribution

To get insights into the relationship between cache replacement algorithms and identified duplicates, we plot the cumulative deduplication ratio of the inter-reference distance (IRD) for LRU, LIRS, ARC and TLE-LRU under different workloads in Fig. 7. Here, the inter-reference distance is defined as the number of data blocks between a fingerprint and its last occurrence. The cache size is set to 80 MB.

For all workloads, ARC only detects duplicates within small IRD values ($IRD < 2 \times 10^6$). LIRS outperforms LRU in each workload setting. Moreover, when the workload has weaker overall temporal locality, LIRS can find more duplicates (with IRD = 5.21×10^6 and 1.59×10^7 for workload B and C, respectively) compared to LRU (IRD = 1.68×10^6 and 1.51×10^6 for workload B and C, respectively). TLE-LRU finds the most duplicates for all



(a) Workload A (L:NL = 3:1) (b) Workload B (L:NL = 2:2) (c) Workload C (L:NL = 1:3)

Fig. 6. Inline deduplication ratio versus cache size for iDedup and SLADE with different cache replacement policies.

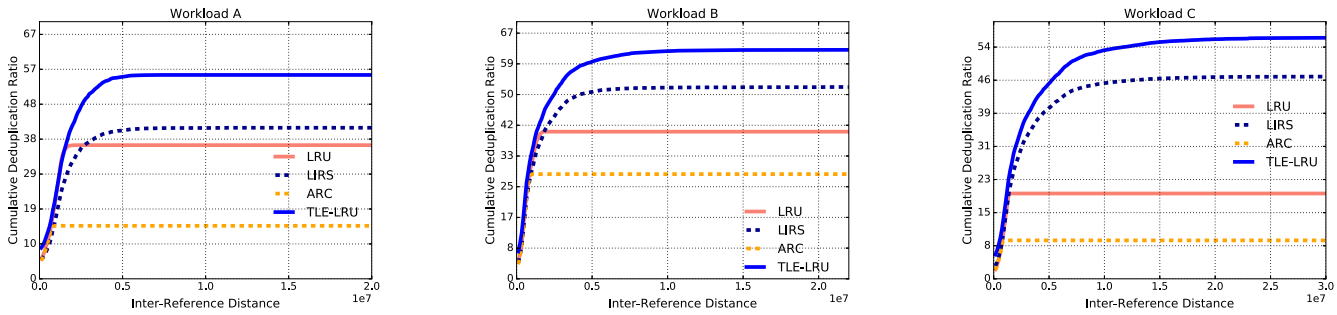


Fig. 7. The cumulative deduplication ratio for different Inter-Reference Distance (IRD) of duplicate blocks.

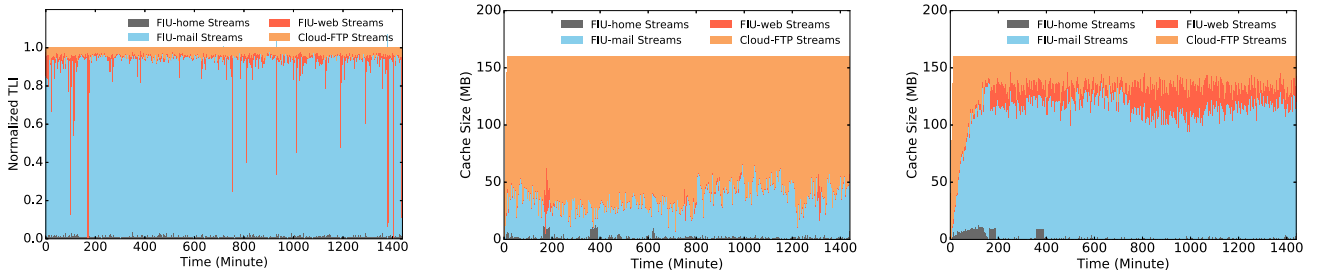


Fig. 8. LDSS estimation accuracy.

workloads as it can differentiate the data streams and avoids caching the fingerprints from streams with weak duplication locality.

7.4 Locality Estimation Accuracy

SLADE improves the efficiency of inline deduplication phase by allocating the fingerprint cache based on temporal locality measured by the *TLI (Temporal Locality Indicator)* of each stream. Fig. 8a shows the normalized observed *TLI* over time for workload B. Here, the cache size is set to 160 MB. The values of *TLI* are normalized. FIU-mail streams show a high *TLI* value, indicating good temporal locality. Without *TLI* estimation, only a tiny portion of the cache is allocated to FIU-mail streams (see Fig. 8b). On the contrary, Cloud-FTP streams with a relatively low *TLI* value are allocated a large portion of the cache space. With *TLI* estimation, as shown in Fig. 8c, the cache space is properly allocated to streams based on their temporal locality. The inline deduplication ratio is, therefore, improved by around 12.53 percent. The improvement shows the effectiveness of *TLI* estimation in SLADE.

The estimation accuracy of *TLI* is critical to the cache efficiency. To demonstrate the sensitivity of TLE-LRU algorithm to the sampling rate, we conducted an experiment on the three workloads under the same cache size configuration. We choose the sampling rate of 5, 10, 15 and 20 percent for evaluation (see Fig. 9). For all the three workloads, the sampling rate of 15 percent works well. For workload C, the sampling rate of 10 percent achieves a similar deduplication ratio with that under the sampling rate of 15 percent. A larger sampling rate is not necessary since the estimation is already accurate.

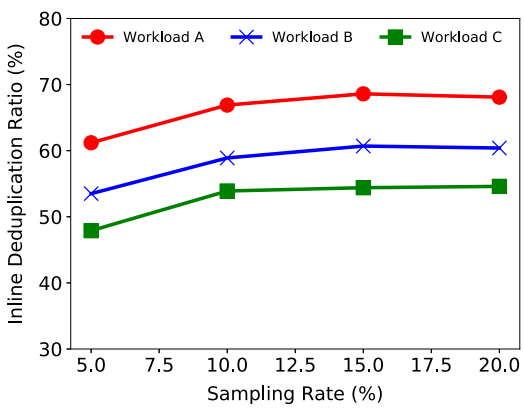


Fig. 9. Sampling rate sensitivity analysis.

7.5 Disk Capacity Requirement

We compare the size of the data written to the disks for SLADE and a pure post-processing deduplication process (e.g., [2]). The data size also represents the maximum disk space required for the incoming data. For SLADE, the inline fingerprint size is set to 160 MB and the TLE-LRU cache replacement policy is used. SLADE significantly reduces the disk capacity requirement by 45.08, 28.29 and 12.78 percent for workload A, B, and C, respectively. The better locality a workload has, the more duplicates can be detected in the inline phase and the more duplicate data writes are eliminated. This clearly shows the benefit of a hybrid deduplication architecture of SLADE as hundred GBs can be saved by only maintaining a 160 MB inline fingerprint cache.

7.6 I/O Latency

Fig. 10 compares the average I/O latency between iDedup and SLADE. Since the original traces are split into blocks, we reconstruct the I/O requests according to the timestamps, the LBA and the length of the trace records as described by Mao et al. [5]. 32 μs are added to the write

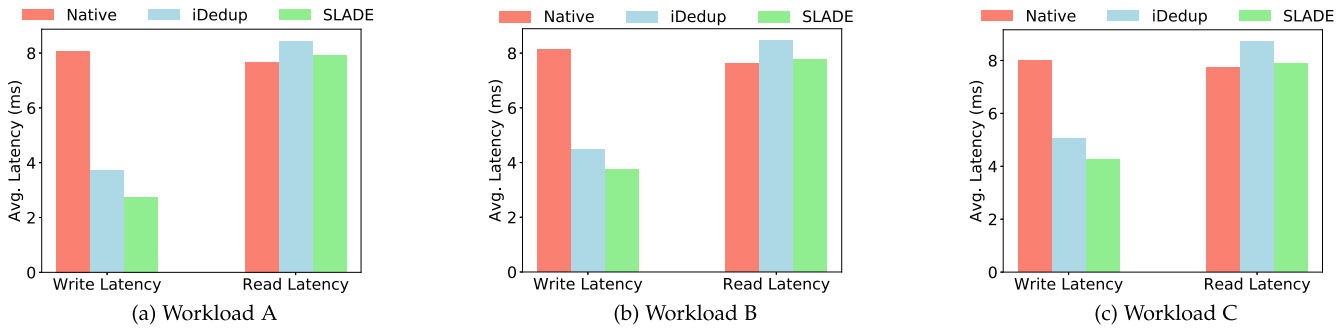


Fig. 10. Average I/O latency of read/write operations of the native system and different deduplication schemes.

latency for each block to simulate the hash computation since the fingerprints have already been computed in the traces. For both iDedup and SLADE, the inline fingerprint cache is set to 320 MB. We disable the read data buffer for a fair comparison.

For workload A, iDedup and SLADE improve the write performance of the native system by 53.9 and 65.8 percent, respectively. This is mainly because of a large number of duplicate data and duplicate I/Os in the workload. The improvement of SLADE comes from both the TLE-LRU caching and the duplicate sequence threshold adjustment. The former identifies more duplicates in the fingerprint cache while the latter can help remove more small duplicate I/Os when random I/Os dominate in a stream. iDedup introduces extra 10.3 percent latency overhead for read operations due to disk fragmentation while SLADE introduces less (3.1 percent) since it gives a higher threshold for the stream which is dominated by long requests. Similarly, SLADE outperforms iDedup by 26.2 and 18.9 percent in write performance for workload B and workload C, respectively. For the read performance, SLADE also shows 7.7 and 10.8 percent improvement over iDedup.

Moreover, the performance impact of the post-processing phase on the I/O latency is minimal mainly due to the following two reasons: First, the post-processing deduplication process is only active during the idle time of the system. Second, the read operations are mostly for the recent data. The experiments show that by exploiting temporal locality of primary workloads, SLADE can improve the write performance of the primary storage system while introducing trivial latency overhead for read operations.

7.7 Fragmentation

In this section, we evaluate the fragmentation of files in storage system caused by deduplication. The length threshold of duplicate block sequence controls the fragmentation in both SLADE and DIODE. Both DIODE and SLADE are able to adjust the threshold dynamically. Fig. 11 shows the threshold change along time for workload A in DIODE and SLADE. For workload A, the inline deduplication ratios for DIODE and SLADE are 57.62 and 68.96 percent, respectively. As one may see, SLADE is able to adjust the threshold for each stream while DIODE uses a global threshold. The FIU-mail and Cloud-FTP have higher thresholds than FIU-home and FIU-web. Since a larger threshold leads to less disk fragmentation, this result shows that SLADE introduces less fragmentation while achieving higher inline deduplication ratio than DIODE.

7.8 Overhead Analysis of TLE-LRU Cache

While SLADE improves the efficiency of primary storage deduplication significantly, it inevitably incurs some overhead. In the inline phase, the overhead is mainly from the TLE-LRU fingerprint cache management. We classify the overhead into two categories: the computational overhead and the memory overhead.

7.8.1 *Computational Overhead*

The computational overhead of SLADE is mainly from the temporal locality estimation. Specifically, it contains two parts: the FFH (Fingerprint Frequency Histogram) generation time and the temporal locality estimation algorithm execution time.

To calculate the FFH, we only need to scan the sampling buffer and count the occurrence of fingerprints to obtain the histogram. Therefore, the time complexity is $O(n)$ where n is the number of samples. Fig. 12 shows the time used for generating the histogram in our current implementation. Here, the sampling rate is 15 percent. We can see that the

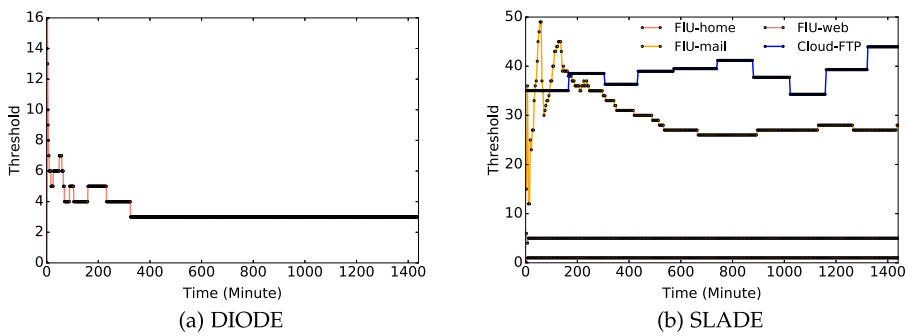


Fig. 11. Threshold versus time for SLADE and DIODE (cache size: 160MB).

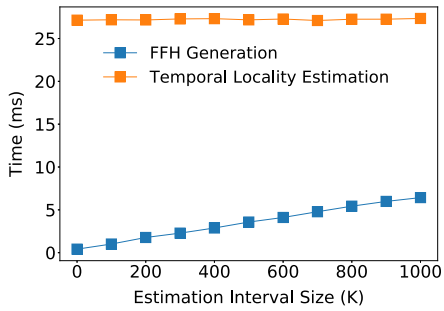


Fig. 12. The computational overhead of TLE-LRU.

histogram calculation of an estimation interval with 1 million blocks takes less than 7 ms.

Estimating the temporal locality of streams is achieved by using the method described in Section 2.2. The core of the estimation is to solve a linear programming problem. The linear programming problem can be solved in $O(n)$ [24] and even constant time [25] when the number of variables d is fixed and the number of constraints is fixed. In our context, the condition is satisfied because duplicates with high frequencies in the sampling buffer will be used in a straightforward way during the estimation and will not be put into the linear programming. Note that the linear programming needs to be done for each stream. For every estimation interval, the temporal locality estimation takes about 26 ms for each stream regardless the estimation interval size (see Fig. 12). The computing overhead is acceptable as the process is performed in the background and does not affect the write performance.

7.8.2 Memory Overhead

The primary memory overhead of SLADE is the sampling buffer. For a sampling rate of 15 percent, the memory overhead is only 4.49 MB (2.81 percent of cache size) even if we choose a large estimation interval factor (e.g., Workload C, 0.6). In practice, for an aggregated stream with strong temporal locality, the memory overhead is low (e.g., 2.25 MB for Workload A and 2.99 MB for Workload B) as the *estimation interval factor* can be set to a small value. Compared with the improvement (19.80 - 25.81 percent) of inline deduplication ratio for the three workloads with 160 MB cache size), the memory overhead of SLADE is low.

7.9 Efficiency of Post-processing Deduplication Phase

For the three workloads, we observed that the majority of duplicates are removed in the inline deduplication. When the inline fingerprint cache is set to 320 MB, around 1.87, 2.59 and 4.25 percent of the total duplicates in the traces were written to the disk for post-processing deduplication. Although the precentages are relatively small, it does not mean that the post-processing phase is unnecessary. The reasons are twofold. First, even only 1 percent duplicates in the traces may require several GBs of storage space to store them. More importantly, the traces only cover a small portion of the data stored on the disks. Although some blocks are not duplicate in the traces, they can still be duplicate when comparing with the blocks which have already been written to the disks. However, since the fingerprints for all data blocks on the disk are not available for the traces, we are not able to evaluate the deduplication ratio for the post-processing deduplication phase.

TABLE 4
Average Throughput and CPU Utilization of the Post-Processing Deduplication Phase for SLADE

Workload	A	B	C
Throughput (MB/s)	30.3	30.7	31.2
Avg. Single Core CPU Utilization	26.8%	26.2%	27.7%

To evaluate the efficiency of the post-processing deduplication phase of SLADE, we examined the throughput, CPU utilization of the system. The in-memory fingerprint cache to prefetch fingerprint is set to 128MB. Fingerprints are prefetched according to the LBA to improve the deduplication throughput.

As shown in Table 4, the CPU utilization for the three workloads are all under 28 percent. The CPU utilization is measured by the average utilization when the post-processing deduplication is triggered. We observed that the throughput of processing the data blocks during post-processing deduplication was around 30 MB per second. For all the workloads, the I/O utilization (the percentage of time used for I/O processing) during the deduplication was around 12 percent. Therefore, the system resource is still available to run some other foreground tasks during the post-processing deduplication. This sustainable throughput allows the post-processing deduplication to be finished within the idle time of the system and the impact on foreground tasks can be minimized.

8 Related Work

Existing work on primary storage deduplication systems can be classified into three categories: *Inline deduplication*, *Post-processing storage deduplication* and *Hybrid inline and post-processing deduplication*.

Inline primary storage deduplication. Most inline primary deduplication exploits the locality of primary workloads to perform non-exact deduplication. iDedup [4] exploits the temporal locality by maintaining an in-memory fingerprint cache. To exploit the spatial locality, iDedup only eliminates duplicates in long sequences of data blocks. POD [5] aims at improving the I/O performance in primary storage systems and mainly performs deduplication on small I/O requests. HANDS [6] uses working set prediction to improve the locality of fingerprints. Koller et al. [14] use a content-aware cache to improve the efficiency of I/O by avoiding the influence of duplicated data. PDFS [10] argues that strong locality may not commonly exist in primary workloads. To avoid the disk bottleneck of storing fingerprint table, a similarity-based partial lookup solution is proposed. Leach [26] exploits temporal locality of workloads using a splay tree. These work do not consider scenarios where the primary storage workloads contain a mix of data streams with different access patterns.

Post-processing primary storage deduplication. Post-processing deduplication performs deduplication during the idle time of primary storage systems. Ahmed El-Shimi et al. [2] propose a post-processing deduplication method in the Windows Server OS. Similar to SLADE, DEDIS [17] is built in the Xen hypervisor and performs deduplication for multiple virtual

machines. The main purpose of the post-processing primary storage deduplication is to avoid the high I/O latency in the write path. However, inline caching is much more cost-effective than post-processing when there are portions of workloads with strong or moderate locality. The contribution of SLADE is to differentiate primary workloads based on their temporal locality and apply different deduplication procedures accordingly.

Hybrid inline and post-processing storage deduplication. RevDedup [27] exploits combining the inline and the post-processing deduplication together in the backup deduplication. Recently, DDFS [28] proposes to use a rate-limiting method to improve the fingerprint indexing while only leaving a modest number of duplicates to be eliminated by the GC mechanism [29] when they become old. The difference between DDFS and SLADE is that DDFS is mainly for backup deduplication while SLADE is designed for primary deduplication. Correspondingly, DDFS exploits spatial locality of the backup workloads while SLADE exploits temporal locality of primary workloads in the inline deduplication phase. For primary storage deduplication, DIODE [22] is a dynamic architecture of inline-offline deduplication. Like ALG-Dedupe [30], DIODE is an application-aware deduplication mechanism. The decision to perform inline deduplication on a file is made based on the file extensions. However, our experiments show that file type information is not sufficient for achieving good inline deduplication performance. The key difference between SLADE and DIODE is that SLADE gives a dynamic locality estimation method to improve inline deduplication performance and reduce the load of the post-processing deduplication process.

Estimating the number of duplicates in a time frame for a data stream is similar to estimating distinct elements in a large set, for which various statistics based methods (e.g., [31], [32]) have been investigated. Fisher et al. [33] describe a method to estimate the number of unknown species given a histogram of randomly sampled species. The theoretical computer science community has been trying to address how to perform the estimation with fewer samples [34], [35], [36], [37]. Recently, this line of work has been extended by Valiant and Valiant [38] to characterize unobserved distributions. They prove that $O(\frac{n}{log(n)})$ samples are sufficient to provide an accurate estimation of the whole dataset, in which n is the size of the whole dataset. Harnik et al. [39] utilize the theory to estimate the deduplication ratio in storage systems. Using the history of workloads to improve the cache efficiency has been explored in flash cache management [40], [41], [42], [43], [44]. These studies use cache admission policies and dynamic cache allocation to address the flash wear-out problem. To the best of our knowledge, SLADE is the first work to use locality estimation to deal with the cache contention problem in inline deduplication for primary storage.

9 Conclusion

We revealed that existing cache algorithms could not cope with the fingerprint cache management well in primary deduplication systems in the cloud. Particularly, we showed that the ghost cache approach which was originally designed for addressing weak locality incurred high memory overhead in fingerprint caching. To address this, we proposed TLE-LRU, a cache replacement algorithm that can estimate the temporal locality of the data streams and prioritize the cache allocation according to the estimation. We built a hybrid deduplication system named SLADE with TLE-LRU. SLADE achieved high inline cache efficiency and reduced the deduplication workload in the post-processing deduplication phase. SLADE improved the inline deduplication ratio by up to 39.70 percent compared with iDedup in our experiments. Meanwhile, SLADE reduced the disk capacity requirement by up to 45.08 percent compared with the post-processing deduplication mechanism in our evaluation.

Acknowledgments

This work is partially supported by the The National Key Research and Development Program of China (2016YFB0200401), by the program for New Century Excellent Talents in University, by National Science Foundation (NSF) China 61402492, 61402486, 61402518, 61379146, by the laboratory pre-research fund (9140C810106150C81001), by the HUNAN Province Science Foundation 2017RS3045. An earlier version of this paper appeared in the 33rd IEEE Symposium on Mass Storage Systems and Technologies (MSST) [45].

References

[1] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 1–14.

[2] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplicationlarge scale study and system design," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 285–296.

[3] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Trans. Storage*, vol. 7, no. 14, pp. 1–20, 2012.

[4] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2012, pp. 1–14.

[5] B. Mao, H. Jiang, S. Wu, and L. Tian, "POD: Performance oriented I/O deduplication for primary storage systems in the cloud," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 767–776.

[6] A. Wildani, E. L. Miller, and O. Rodeh, "Hands: A heuristically arranged non-backup in-line deduplication system," in *Proc. IEEE 29th Int. Conf. Data Eng.*, 2013, pp. 446–457.

[7] J. An and D. Shin, "Offline deduplication-aware block separation for solid state disk," in *Proc. 11th USENIX Conf. File Storage Technol.*, 2013, pp. 1–2.

[8] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," in *Proc. Data Compression Conf.*, 2011, pp. 393–402.

[9] V. Tarasov, et al., "Dmdedup: Device mapper target for data deduplication," in *Proc. Ottawa Linux Symp.*, 2014, pp. 83–95.

[10] H. Yu, X. Zhang, W. Huang, and W. Zheng, "PDFS: Partially dedupped file system for primary workloads," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 863–876, Mar. 2017.

[11] J. Kaiser, T. Süß, L. Nagel, and A. Brinkmann, "Sorted deduplication: How to process thousands of backup streams," in *Proc. 32nd Symp. Mass Storage Syst. Technol.*, 2016, pp. 1–14.

[12] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 30, no. 1, pp. 31–42, 2002.

[13] N. Megiddo and D. S. Modha, "ARC: A self-tuning, low overhead replacement cache," in *Proc. 2nd USENIX Conf. File Storage Technol.*, 2003, vol. 3, pp. 115–130.

[14] R. Koller and R. Rangaswami, "I/O deduplication: Utilizing content similarity to improve I/O performance," *ACM Trans. Storage*, vol. 6, no. 13, pp. 1–26, 2010.

[15] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, 2014, Art. no. 2.

[16] J. Rhoton, de Jan Clercq, and F. Novak, *OpenStack Cloud Computing*. Boca Raton, FL, USA: Recursive Press, 2014.
[17] J. Paulo and J. Pereira, "Efficient deduplication in a distributed primary storage infrastructure," *ACM Trans. Storage*, vol. 12, no. 4, pp. 20:1–20:35, 2016.
[18] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "CacheDedup: In-line deduplication for flash caching," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 301–314.
[19] Leveldb: A fast and lightweight key/value database library by google. (2014). [Online]. Available: https://github.com/google/leveldb, Accessed on: 04-Dec-2017.
[20] P. Valiant and G. Valiant, "Estimating the unseen: Improved estimators for entropy and other properties," in *Proc. Int. Conf. Adv. Neural Inform. Process. Syst.*, 2013, pp. 2157–2165.
[21] M. Al-Kateb, B. S. Lee, and X. S. Wang, "Adaptive-size reservoir sampling over data streams," in *Proc. IEEE 19th Int. Conf. Scientific Statist. Database Manag.*, 2007, pp. 22–22.
[22] Y. Tang, J. Yin, S. Deng, and Y. Li, "DIODE: Dynamic inline-offline de duplication providing efficient space-saving and read/write performance for primary storage systems," in *Proc. IEEE 24th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2016, pp. 481–486.
[23] Z. Sun, et al., "A long-term user-centric analysis of deduplication patterns," in *Proc. 32nd Symp. Mass Storage Syst. Technolo.*, 2016, pp. 1–7.
[24] N. Megiddo, "Linear programming in linear time when the dimension is fixed," *J. ACM*, vol. 31, no. 1, pp. 114–127, 1984.
[25] N. Alon and N. Megiddo, "Parallel linear programming in fixed dimension almost surely in constant time," in *Proc. 31st Annu. Symp. Found. Comput. Sci.*, 1990, pp. 574–582.
[26] B. Lin, S. Li, X. Liao, J. Zhang, and X. Liu, "Leach: An automatic learning cache for inline primary deduplication system," *Frontiers Comput. Sci.*, vol. 8, no. 2, pp. 175–183, 2014.
[27] Y.-K. Li, M. Xu, C.-H. Ng, and P. P. Lee, "Efficient hybrid inline and out-of-line deduplication for backup storage," *ACM Trans. Storage*, vol. 11, no. 1, 2015, Art. no. 2.
[28] Y. Allu, F. Douglis, M. Kamat, P. Shilane, H. Patterson, and B. Zhu, "Backup to the future: How workload and hardware changes continually redefine data domain file systems," *Comput.*, vol. 50, no. 7, pp. 64–72, 2017.
[29] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. Botelho, "The logic of physical garbage collection in deduplicating storage," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 29–44.
[30] Y. Fu, H. Jiang, N. Xiao, L. Tian, F. Liu, and L. Xu, "Application-aware local-global source deduplication for cloud backup services of personal storage," *IEEE Trans. Parallel Distrib. Syst.*, vol. 5, no. 25, pp. 1155–1165, May 2014.
[31] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes, "Sampling-based estimation of the number of distinct values of an attribute," in *Proc. 30th Int. Conf. Very Large Data Bases*, 1995, vol. 95, pp. 311–322.
[32] C. X. Mao and B. G. Lindsay, "Estimating the number of classes," *Ann. Statist.*, vol. 35, pp. 917–930, 2007.
[33] R. A. Fisher, A. S. Corbet, and C. B. Williams, "The relation between the number of species and the number of individuals in a random sample of an animal population," *J. Animal Ecology*, vol. 12, pp. 42–58, 1943.
[34] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Sampling algorithms: Lower bounds and applications," in *Proc. 33rd Annu. ACM Symp. Theory Comput.*, 2001, pp. 266–275.
[35] T. Batu, S. Dasgupta, R. Kumar, and R. Rubinfeld, "The complexity of approximating the entropy," *SIAM J. Comput.*, vol. 35, no. 1, pp. 132–150, 2005.
[36] P. Valiant, "Testing symmetric properties of distributions," *SIAM J. Comput.*, vol. 40, no. 6, pp. 1927–1968, 2011.
[37] S. Guha, A. McGregor, and S. Venkatasubramanian, "Streaming and sublinear approximation of entropy and information distances," in *Proc. 17th Annu. ACM-SIAM Symp. Discrete Algorithm*, 2006, pp. 733–742.
[38] G. Valiant and P. Valiant, "Estimating the unseen: An n/log (n)-sample estimator for entropy and support size, shown optimal via new CLTS," in *Proc. 43rd Annu. ACM Symp. Theory Comput.*, 2011, pp. 685–694.
[39] D. Harnik, E. Khaitzin, and D. Sotnikov, "Estimating unseen deduplication-from theory to practice," in *Proc. 14th USENIX Conf. File Storage Technol.*, 2016, pp. 277–290.
[40] 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.
[41] J. Yang, N. Plasson, G. Gillis, and N. Talagala, "HEC: Improving endurance of high performance flash-based cache devices," in *Proc. 6th Int. Syst. Storage Conf.*, 2013, pp. 1–11.
[42] 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. 8, pp. 1–24, 2016.
[43] J. Liu, Y. Chai, X. Qin, and Y. Xiao, "PLC-cache: Endurable SSD cache for deduplication-based primary storage," in *Proc. 30th Symp. Mass Storage Syst. Technol.* pp. 1–12, 2014.
[44] R. Koller, A. J. Mashtizadeh, and R. Rangaswami, "Centaur: Host-side SSD caching for storage performance control," in *Proc. IEEE Int. Conf. Autonomic Comput.*, 2015, pp. 51–60.
[45] H. Wu, C. Wang, Y. Fu, S. Sakr, L. Zhu, and K. Lu, "Hpdedup: A hybrid prioritized data deduplication mechanism for primary storage in the cloud," in *Proc. 33rd Symp. Mass Storage Syst. Technol.*, 2017, pp. 1–14.

Huijun Wu received the ME degree in computer science from the National University of Defense Technology, in 2015. He is currently working toward the PhD degree majoring in computer science with the University of New South Wales, Australia as well as Data61, CSIRO. His current research interests include data deduplication, data compression for storage systems.

Chen Wang received the PhD degree from Nanjing University. He is a senior research scientist with Data61, CSIRO. His research is primarily in distributed and parallel computing area. He published more than 60 papers in major journals and conferences. He has industrial experience. He developed a high-throughput event system and a medical image archive system used by many hospitals and medical centers in the USA.

Yinjin Fu received the PhD degree in computer science from the National University of Defense Technology, China, in 2013. He is an assistant professor in computer science with the Army University of Engineering, China. He visited the University of Nebraska-Lincoln from 2010 to 2012. His research interests include data deduplication and file systems. He has more than 30 publications in major journals and conferences. He is a member of the IEEE.

Sherif Sakr received the PhD degree in computer and information science from Konstanz University, Germany, in 2007. He is a professor in the Department of Health Informatics, King Saud bin Abdulaziz University for Health Sciences, Saudi Arabia. He is also affiliated with the University of New South Wales (Australia) and with the Data61, CSIRO. He is a senior member of the IEEE and an IEEE distinguished speaker.

Kai Lu received the graduated degree from the National University of Defense Technology, and the PhD degree in computer science and technology from the National University of Defense Technology, in 1999. He is a professor and deputy dean of the College of Computer Science, National University of Defense Technology, China. He is the associated chief designer of the Tianhe-1 (MilkyWay-1) and the Tianhe-2 (MilkyWay-2) supercomputer systems which ranked the 1st in TOP500 from June 2013 to June 2016, respectively.

Liming Zhu received the PhD degree in software engineering from the University of New South Wales, in 2007. He is the research director of software and computational systems research program with Data61, CSIRO. He holds conjoint professor positions with the University of New South Wales (UNSW) and University of Sydney (UTS). He has published more than 120 peer-reviewed conference and journal articles. His research interests include dependable systems and data analytics infrastructure.