Park D, Fan Z, Nam YJ *et al.* A lookahead read cache: Improving read performance for deduplication backup storage. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 32(1): 26–40 Jan. 2017. DOI 10.1007/s11390-017-1680-8

A Lookahead Read Cache: Improving Read Performance for Deduplication Backup Storage

Dongchul Park, Ziqi Fan, Young Jin Nam, and David H. C. Du, *Fellow, IEEE*

Department of Computer Science and Engineering, University of Minnesota–Twin Cities, Minneapolis, MN 55455, U.S.A.

E-mail: {park, fan, youngjin, du}@cs.umn.edu

Received March 11, 2016; revised September 18, 2016.

Abstract Data deduplication (dedupe for short) is a special data compression technique. It has been widely adopted to save backup time as well as storage space, particularly in backup storage systems. Therefore, most dedupe research has primarily focused on improving dedupe write performance. However, backup storage dedupe read performance is also a crucial problem for storage recovery. This paper designs a new dedupe storage read cache for backup applications that improves read performance by exploiting a special characteristic: the read sequence is the same as the write sequence. Consequently, for better cache utilization, by looking ahead for future references within a moving window, it evicts victims from the cache having the smallest future access. Moreover, to further improve read cache performance, it maintains a small log buffer to judiciously cache future access data chunks. Extensive experiments with real-world backup workloads demonstrate that the proposed read cache scheme improves read performance by up to 64.3%

Keywords deduplication, dedupe, read cache, backup

1 Introduction

Today's digital data explosion has propelled data deduplication (dedupe for short) into the spotlight. Over 80% of companies are considering dedupe technologies[1]. In addition, due to the home digital data explosion, personal consumers have increasingly adopted personal backup storage systems such as home network attached storage (NAS) for reliable data storage and backup[2]. Most of these systems are equipped with their own backup systems. It has been reported that about 75%, or even as much as 80%, of digital data is duplicated[3-4]. Hence, deduplication technology is of great importance for both industry and in the home.

Data dedupe is a specialized technique to eliminate duplicate data so that it retains only one unique data copy on storage. It replaces redundant data with a pointer to the unique data afterwards. Today, dedupe technologies are widely deployed to save cost, particularly in secondary storage systems for data backup or archiving. Therefore, most dedupe research has primarily focused on dedupe write performance improvement: how well (or how efficiently) dedupe storage systems can detect and eliminate duplicate data chunks. This includes index optimization and caching, efficient data chunking, compression, and data container design[5-11].

On the other hand, dedupe read performance has not attracted considerable attention because read operations are rarely invoked during system backup or archiving. However, when it comes to system recovery, it is a totally different story. Long term digital preservation (LTDP) communities also emphasize read performance importance in dedupe storage[12]. Moreover, some primary storage systems have started to provide dedupe technologies where the number of read operations even exceeds the number of write operations[13-14]. Although read performance is also a crucial dedupe storage consideration, relatively little effort has been devoted to this problem.

A typical dedupe read operation processes a data chunk (generally, 4 KB∼8 KB) in secondary dedupe

Regular Paper

This work is partially supported by the National Science Foundation Awards of USA under Grant Nos. 121756, 1305237, 142191 and 1439622.

storage as follows: first, the dedupe storage system identifies the data container ID which stores the corresponding data chunks to read. Then, it looks up the container (generally, 2 or 4 MB) in the read cache. Once hitting the cache, it can directly read the chunks from the cache. Otherwise, it fetches one whole container from the underlying storage, allowing it to read the corresponding data chunks in the container. In general, more duplicate data chunks (i.e., higher dedupe rate) give rise to higher data fragmentation, which is a root cause of dedupe read performance degradation. This performance decrease stems from the fact that only partial data chunks are accessed in a container and the remainder (the majority) in the container is unavoidably fetched into memory with no access.

To resolve this limitation, allowing duplicate data (that is, selective duplication) is a possible solution to improve the dedupe read performance[1,15-16]. This turns random data accesses into sequential accesses, increasing cache utilization due to higher data container spatial localities. However, this inevitably hurts dedupe write throughput (i.e., dedupe rate) due to more duplicate data chunks.

This paper proposes a novel dedupe storage read cache design for a backup application named a lookahead read cache. This proposed design is a pure read cache design with no need to sub-optimize the dedupe rate. Hence, the dedupe system can achieve both a complete dedupe and fast read performance. Moreover, it can be efficiently deployed to existing dedupe systems with minimum changes.

The key idea is to exploit future data chunk access patterns. In general, for backup applications, read sequences are identical to write sequences in dedupe storage[16-17]. Inspired by this special characteristic, the read cache design can exploit future read access patterns during dedupe processes. Consequently, the proposed scheme outperforms a widely adopted cache algorithm in backup storage systems, the least recently used (LRU) scheme. The proposed lookahead read cache maintains a future reference count for each data container. Unlike the LRU, it evicts a data container with the smallest future reference count from the cache.

Furthermore, based on real backup dataset analysis observations, employing a small log buffer that keeps small portions of future reference data chunks further improves the lookahead read cache design. The main contributions of this paper are as follows.

• *Future Access-Based Read Cache Design.* It maintains access information for future read references during dedupe (i.e., write) processes. Thus, the proposed design evicts from the read cache a victim with a smallest future reference count.

• *Extended Cache Design with a Log Buffer.* A lookahead read cache assigns a portion of a read cache space to a log buffer, which can effectively maintain future access chunks on the basis of our hot data identification scheme adopting the counting bloom filter and multiple hash functions.

• *Extensive Dataset Workload Analysis.* The proposed design is fundamentally inspired by the workload analysis of diverse real backup datasets.

The remainder of this paper is organized as follows. Section 2 presents the background of the work. Section 3 explains the design and operations of the proposed cache scheme. Section 4 provides a variety of experimental results and analyses. Section 5 discusses related work addressing especially the dedupe read performance problem. Finally, Section 6 concludes this work.

2 Background

This section first describes data deduplication techniques and then discusses a read performance problem in the existing data deduplication systems.

2.1 Data Deduplication

Data deduplication can efficiently eliminate duplicates from a large number of data streams and it has already become a key capability of current commercial backup storage systems[18]. Fig.1 shows a typical dedupe process that mainly consists of chunking and dedupe logic. A typical dedupe process begins with dividing a given data stream into smaller chunks with variable/fixed lengths, and computes a hash value of each chunk (digested chunk information) with a cryptographic hash function (such as SHA-1). Generally, dynamic chunking (variable-length chunks) with fingerprinting[20] or its variants outperforms static chunking (fixed-length chunks) for data backup applications[7], where an average chunk size ranges from 4 KB to 8 KB[7-8,21]. However, a static chunking technique works well for some applications such as VM disk images[22]. A hash index table, also known as a type of key-value store[23], is required to effectively associate a large number of hash values with their storage locations. Only if the hash value of a chunk is not found in the hash index table (assuming all hash values are kept in the table), the chunk (i.e.,

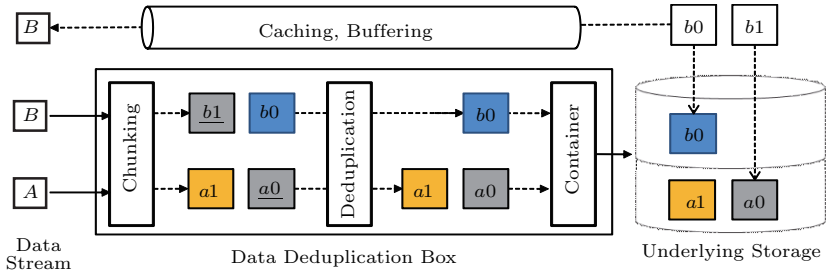


Fig.1. Typical data deduplication process[19]. A and B are data streams. a and b are data chunks of A and B respectively. For instance, data stream A is composed of $a1$ and $a0$.

unique chunk) is allowed to be written to the underlying storage. Otherwise, the chunk (i.e., shared/duplicate chunk) is ignored. The collision probability of a cryptographic hash function is very low enough compared with the soft error rate of the storage system[24]. The unique chunks are not directly written to the underlying storage because the chunk sizes are not large enough to achieve high write performance. Instead, they are organized into a fixed-sized container[8], where the container size is significantly bigger than the average chunk size. The unique chunks are initially buffered into the in-memory container that is allocated for each data stream. Once the in-memory container is full of chunks, it is flushed to the storage. To read a chunk from storage, the entire corresponding container must be first read and then the small chunk in the container can be accessed. Directly reading a small chunk is not allowed.

Recent dedupe research has primarily focused on maximizing the efficient duplication detection by using better chunking[25-27], optimized hash indexing[7,23], locality-preserving index caching[8], and various bloom filters[8,28]. Moreover, good write performance has been its primary goal by compressing unique chunks and performing (fixed length) large writes (2 or 4 MB) through containers[5,8] or similar structures[6].

2.2 Read Performance in Data Deduplication

Dedupe storage read performance (mainly throughput), such as for backup systems, has not been spotlighted since it was widely accepted that read operations are executed less frequently than write operations in such systems. However, read performance becomes extremely critical when restoring entire systems from crashes[8,19]. Higher read performance can significantly save the recovery time, thereby providing higher system availability. Thus, a system may require guaranteed dedupe storage read performance in order to meet a target system recovery time from a crash (e.g., service-level agreement)[29]. In addition, since the dedupe storage has limited storage capacity, it occasionally needs to stage the deduped data in the underlying storage to archival storage media such as a virtual tape. This requires reconstructing the original data streams because archival storage operations are typically stream-based. In fact, this staging frequency is remarkably higher than user-triggered data retrieval frequency.

As duplicate data increases, dedupe storage read data performance generally decreases because reading a data stream requires retrieving both unique and shared chunks, whereas write throughput increases because it stores only unique chunks. Shared chunks are likely to have been physically distributed over different data containers in the underlying storage. This is called chunk fragmentation.

Fig.2 illustrates an example of this chunk fragmentation. Assume that the chunks marked with 10, 11, 12, and 13 are unique chunks. On the contrary, the chunks marked with 200 and 201, 300, and 400 are shared chunks duplicated with the chunks stored in containers 20, 30, and 7, respectively. It is also simply assumed that each container has four chunks. If all eight incoming chunks are unique, the data stream read request ideally requires retrieving only two containers from storage. However, in this example, it requires accessing four different containers. Obviously, this degrades read performance. In reality, the actual read performance significantly drops because an even smaller container portion is only accessed.

3 Lookahead Read Cache

This section first presents the extensive analysis of real backup datasets. It then explores the proposed lookahead read cache design by adopting both a counting bloom filter and a sliding window. To improve

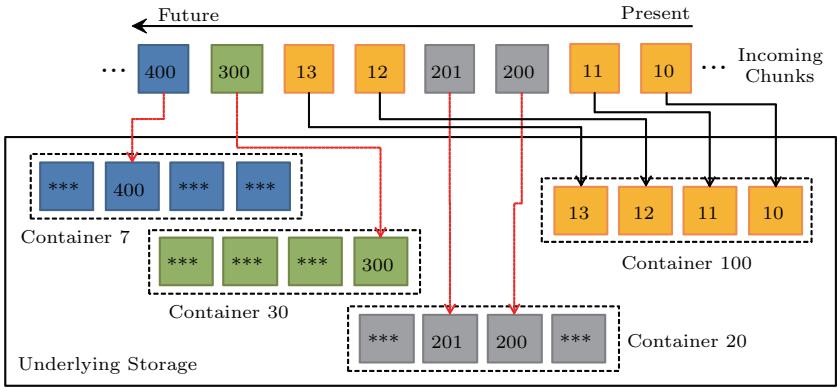


Fig.2. Chunk fragmentation over different containers caused by data deduplication.

performance further, it is extended by employing a log buffer and a hot data identification scheme.

3.1 Rationale and Assumptions

In general, as more incoming data stream duplicates are eliminated, subsequent read performance stands in marked contrast to the initial good write performance due to the higher likelihood of shared data fragmentation[1]. This is a fundamental challenge in the tradeoff between dedupe storage read performance and write performance. To address this read performance problem, a novel read cache design leveraging future access information is proposed. In dedupe storage for backup or archive data, a read access sequence is highly likely to be identical to its write sequence. Based on this key observation, during each dedupe process, the proposed scheme records write access metadata information for future read access. This enables a lookahead read cache to exploit future read references. It is assumed that each data chunk has a variable size and a data container containing many data chunks (generally, 200∼300 chunks) is a basic read unit.

Please note that dedupe storage systems apply this proposed read cache design to each incoming data stream. Moreover, enterprise dedupe storage production systems can concurrently have tens to several hundred of incoming dedupe data streams. Therefore, for each stream to have its own cache, it is important to keep the per-stream cache memory size in the range of megabytes[30].

3.2 Dataset Analysis

An extensive analysis of six real backup datasets is made. It is observed that considerable portions of data are, in general, duplicate for each version of backup datasets. For instance, as Table 1 presents, the backup dataset 2 (i.e., ds-2) has all unique data in the first backup version (i.e., Ver-1). However, in the second backup version, a majority of the data are duplicated (72%). This implies a typical dedupe read cache can be poorly utilized because only a small number of data chunks in a data container are accessed.

Table 1. Average Dedupe Gain Ratio (DGR) in Successive Version of Each Backup Dataset

Dateset	Ver-1	Ver-2	Ver-3	Ver-4	Ver-5	DGR (%)
ds-1	99.9	3.5	6.9	5.6	31.2	29
ds-2	100.0	28.0	24.7	14.9	20.6	37
ds-3	99.6	95.2	97.7	97.3	96.6	97
ds-4	90.5	55.4	63.6	20.8	20.6	50
ds-5	84.1	3.3	2.5	11.9	2.6	20
ds-6	54.4	22.4	–	–	–	38

Note: DGR represents the ratio of a data saving size to an original data size.

Fig.3 exhibits the distributions of the number of accessed data containers with respect to the percentage of accessed chunks in the shared container for each backup dataset. That is, Fig.3 observes how many data chunks in shared containers are accessed when duplicate data chunks are requested for reads. When it reaches 5% (note: logarithmic scale for X-axis in Fig.3(a)), it can accommodate most data containers. This implies that for many data containers, only less than 5% of data chunks in each container are accessed. For both ds-4 and ds-5, even more than 90% of data containers are accessed only less than 5% (Fig.3(b)). Therefore, based on these observations, the proposed design adopts 5% as an initial hot threshold value and the impact of various hot threshold values is explored in the experiment section.

Fig.4 shows container access patterns for the six dedupe backup datasets. The X-axis represents a

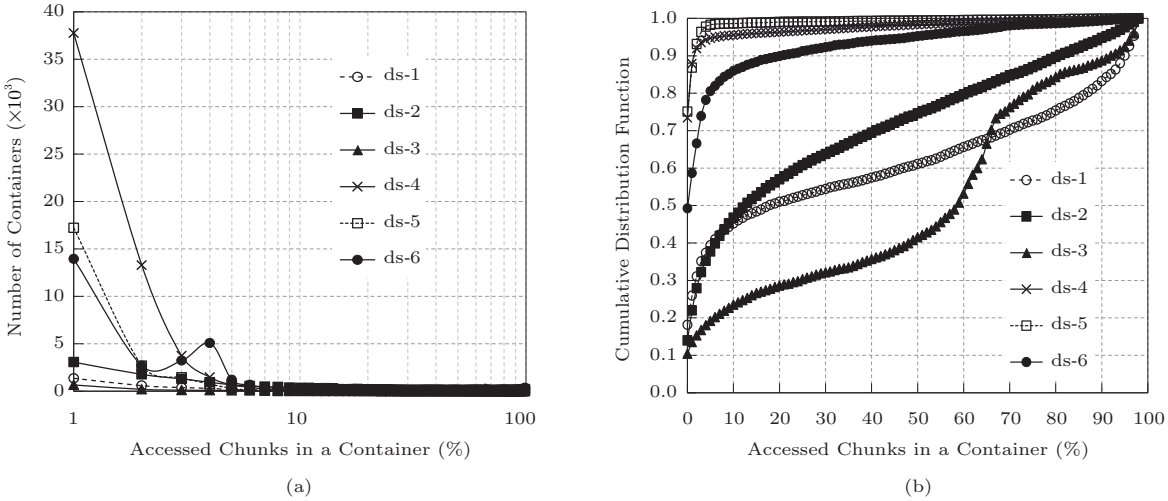


Fig.3. Distributions of the number of accessed containers for six real backup datasets. X-axis represents the percentage of accessed chunks in a container. (a) Container access distribution. (b) CDF (cumulative distribution function).

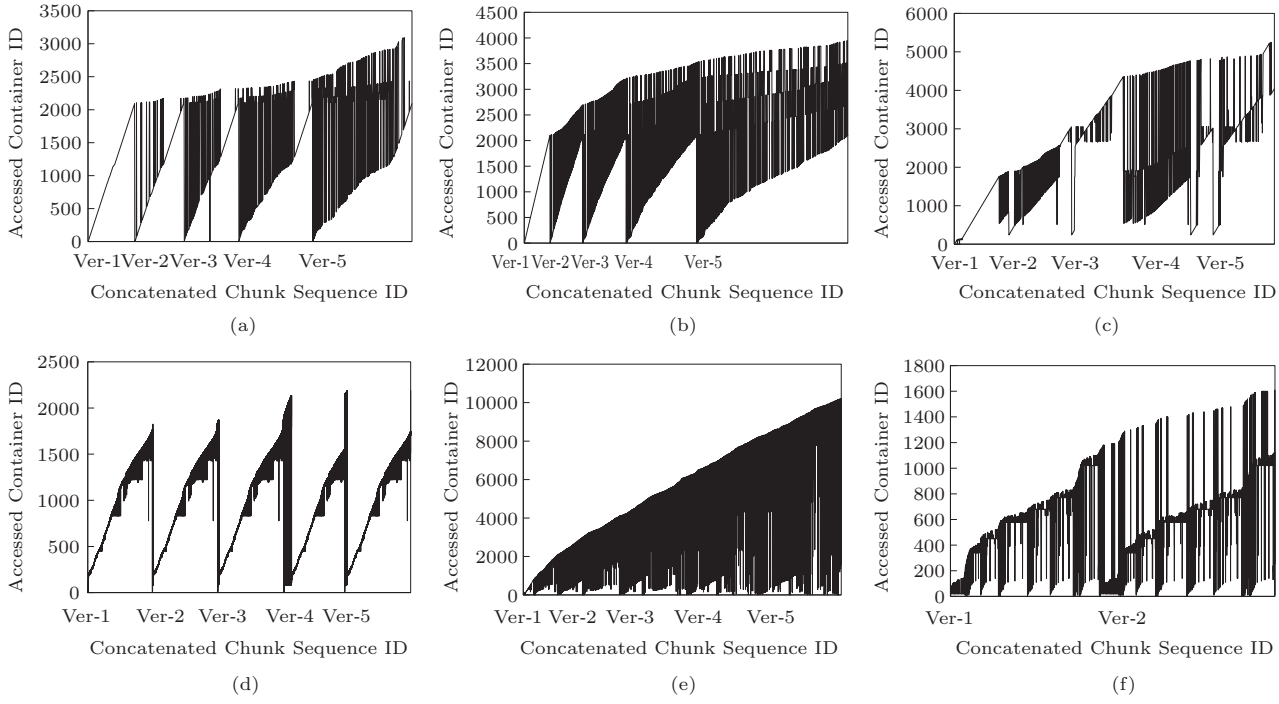


Fig.4. Container access patterns of six datasets. (a) ds-1. (b) ds-2. (c) ds-3. (d) ds-4. (e) ds-5. (f) ds-6.

chunk sequence ID of successive versions of each backup dataset. The Y-axis represents accessed-container IDs storing the data chunks in the X-axis. The container ID starts from 0. For example, if a dataset includes many unique data chunks, the accessed container ID increases linearly, like the first version (Ver-1) of the dataset ds-1: there are no fluctuations within the Ver-1 section in Fig.4(a), whereas if a dataset contains many duplicates, the chart fluctuates due to fragmented accesses of previous containers (Ver-2∼Ver-5 in Fig.4(a)).

Four datasets (ds-1∼ds-4) also show similar data access patterns: most data chunks are unique in the initial backup version (Ver-1) and duplicate chunks increase for each successive backup dataset (Fig.4(a)∼Fig.4(d)). In fact, this is a very typical characteristic of most reverse-referencing backup datasets. That is, the newest backup is made up of pointers that point backwards in time to older backups. Thus, many fluctuations are observable in newer backups. On the other hand, the other two datasets (ds-5 and ds-6) exhibit

different access patterns: there are many duplicates even in the initial version of backup datasets (Ver-1). Thus, many vertical lines are observable, even in Ver-1 as well as successive versions of backup datasets (Fig.4(e) and Fig.4(f)). Although many duplicates are observable, even in the oldest backup version of these two datasets, more duplicates (i.e., fluctuations) still exist in newer backups. This implies these are also reverse-referencing backup datasets. Generally, a reverse-referencing approach exhibits a lower read performance in newer backup datasets due to higher data chunk fragmentation as a consequence of more deduplicated data chunks. This can be observed in Subsection 4.2.

3.3 Architectural Overview

Fig.5 illustrates an overview of the proposed backup storage base read cache design. The proposed scheme consists of three key components: 1) a future sliding window, 2) a hot data identification scheme, and 3) a read cache. The sliding window can be regarded as a lookahead window to estimate future chunk read accesses. The data chunk in the sliding window's tail position is always considered as a current data chunk to read. After reading the current chunk, the window takes a slide toward the future direction by one chunk. Based on this sliding window scheme, the hot or cold decision is made. The hot data identification scheme maintains a future reference counter for each data container. When any data chunk comes into the head position in the window, its reference count of the corresponding data container is incremented by 1 (note: a data container is a basic unit of the chunk read). On the contrary, if any data chunk is evicted from the tail position in the window, the reference count is decreased by 1. If a total reference count of a data container storing current access chunk is greater than a predefined threshold value (e.g., 5%) within the window, the corresponding container is classified as a hot container; otherwise, a cold container. A more detailed architecture of this hot data identification scheme is described in Subsection 3.4. Lastly, the proposed read cache stores the accessed containers and does not adopt the existing cache algorithms such as LRU. Instead, when it is full of data containers fetched from underlying storage, it selects a container with the smallest reference count as a cache eviction victim.

3.4 Hot Data Identification Scheme

Hot data identification plays an important role in the read cache design. Since it is invoked each time when a data chunk is accessed, it must achieve low computational overhead and small memory consumption[31]. To meet these requirements, both a 4-bit single counting bloom filter and multiple hash functions are adopted as Fig.6 shows. Please note the multiple bloom filter-based hot data identification scheme proposed by Park and Du[31] employs multiple independent bloom filters and hash functions to capture recency as well as frequency.

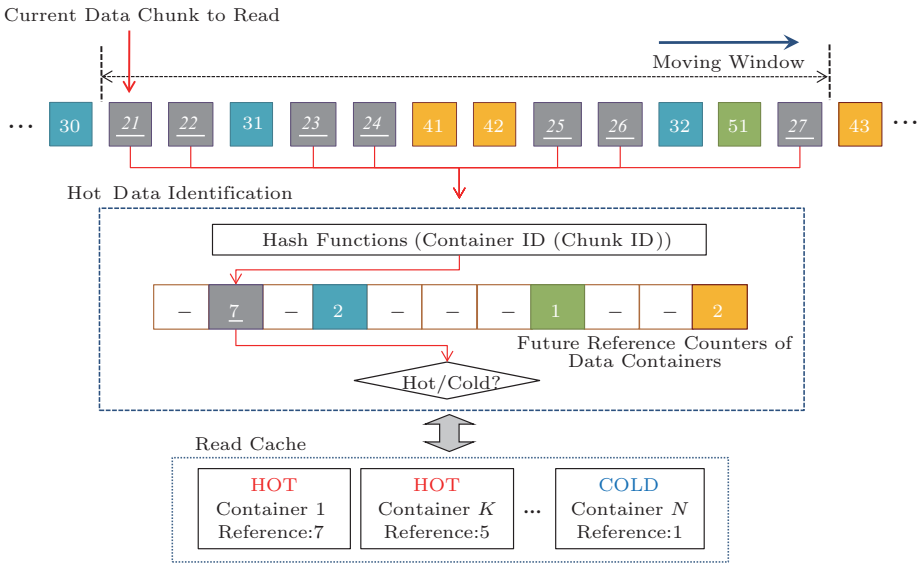


Fig.5. Base read cache design architecture. It is assumed that each data chunk with same color belongs to the same data container.

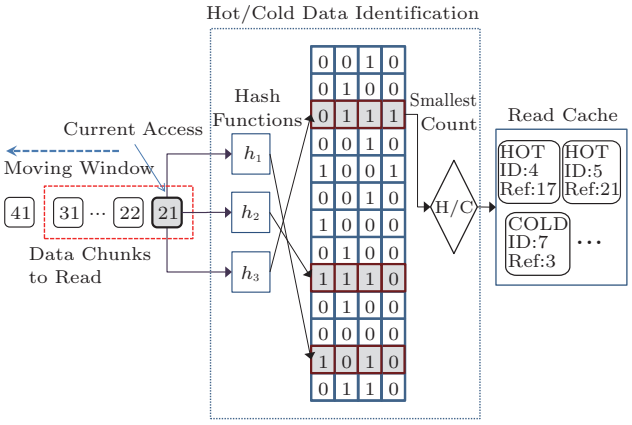


Fig.6. Architecture of the hot data identification scheme. H/C: Hot/Cold. Ref: Reference.

The aforementioned container reference counters are implemented by the counting bloom filter. This hot data identification scheme works as follows: whenever a data chunk comes in the sliding window to the head position, the chunk ID is converted into its container ID and then the container ID is fed to each hash function. Multiple hash functions are employed to reduce a false identification rate in the bloom filter. Each hash value corresponds to its bit position in the bloom filter. Finally, each reference counter is increased by 1. Similarly, the outgoing data chunk from the sliding window decreases each counter by 1 accordingly.

To identify a current access chunk based on these basic operations, its container ID is fed to multiple hash functions. Then, the scheme checks its corresponding reference counters in each bloom filter bit-position. Due to potential hash collisions, the scheme always chooses a smallest reference count for each count value. If it is greater than a predefined threshold, the data container is classified as hot; otherwise it is classified as cold.

3.5 Base Design

Initially, a small (8 MB $= 4 \times 2$ MB container) read cache space is assigned. This is a common and reasonable size in dedupe storage systems which run multiple dedupe processes (usually hundreds of dedupe processes) concurrently.

Overall working processes are described as follows: whenever any data chunk is accessed for read, its container ID is first identified. Next, the container is ready to be stored in the cache. If the data container read cache is full, a victim selection is required. The proposed scheme selects a container with the smallest future reference count as a victim. However, the reference counters of each data container in the cache may change over time because the window continues moving forward. Therefore, the reference counts need updating. That is, when the sliding window moves beyond a chunk, the scheme decreases the reference counters of the data container in the bloom filter by 1. If the data container exists in the cache, it also decreases the corresponding reference counter by 1. Similarly, when a chunk comes into the sliding window, it increases the reference counters in the bloom filter (and in the cache if any) by 1. This update produces negligible overheads because not only are there only a few (basically, four) containers in the cache, but their reference counters need updating only if the corresponding containers reside in the cache.

Even though this algorithm is simple, it considerably improves dedupe read performance and outperforms the widely adopted LRU cache algorithm. Unlike the extended design, this base design utilizes the hot data identification scheme mainly for its efficient reference count management, not for hot data identification.

3.6 Extended Design

This extended read cache design is inspired by dataset workload analysis observations. As in Fig.7, a portion of the read cache space is assigned to a log buffer to more judiciously exploit future access information. The key idea of this extended design is to maintain a small buffer for logging future access chunks before a container is evicted from the read cache in accordance with the hot data identification scheme. This log buffer is managed by a circular queue. Before eviction, the extended design attempts to classify the victim container as either hot or cold by using the hot data identification scheme. If a victim container is identified as cold (that is, only a small number of chunks in the container will be accessed in the near future), its remaining future access chunks within the window are stored in the log buffer. It does not store already-accessed data chunks. Since it employs 5% as a hot threshold value, the maximum amount of data chunks to be logged is at most 5% of the total data chunks (typically 10∼20 chunks) in the victim container. However, in most cases, the amount of logging chunks will be less than the threshold because the window moves forward over time and only the remaining future access chunks in the window are stored to the log buffer. On the other hand, if a victim is identified as hot, the scheme simply discards the victim container without any logging

because it can still achieve high container utilization with this hot container in the near future access. This policy is based on the fact that if it stores many remaining future access chunks (in the hot container) into the log buffer, they also lead to the unnecessary eviction of many logged chunks in the buffer.

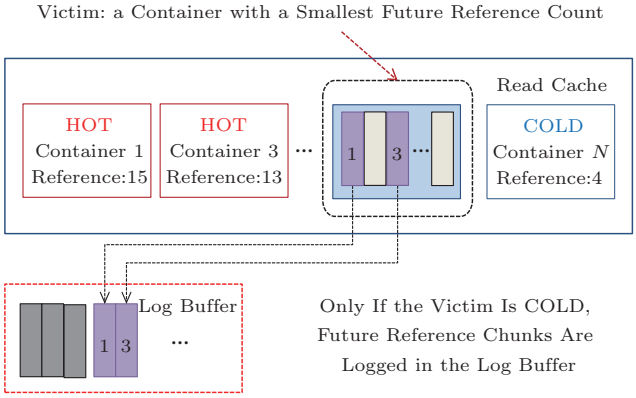


Fig.7. Architecture of our extended read cache design.

This extended read cache design may be comparable to the existing ARC (adaptive replacement cache) design[32] in the sense that evicted data is given a second chance by not discarding the evicted data at once. However, both schemes have significant differences. Specifically, ARC maintains ghost caches for both recency cache and frequency cache. Each ghost cache tracks the history of recently evicted cache entries and is used to adapt to recent resource usage change. Please note that ARC ghost caches only contain metadata, not resource data. That is, if an entry is evicted from either the recency cache or the frequency cache into a ghost cache, its data is discarded. On the other hand, our log buffer keeps evicted data (not the metadata) from a read cache and is not used to adjust a read cache size. Moreover, unlike ARC, even if any data in the log buffer hits, it is not promoted to a read cache. Our read cache is not subdivided into two caches such as recency cache and frequency cache. Finally, our log buffer caching policy is also totally different: our log buffer only stores parts of data evicted from the read cache (i.e., only remains future access chunks in a cold container). Not all data evicted from the read cache are stored in the log buffer. Therefore, the proposed log buffer design is significant different from the ARC cache scheme.

4 Performance Evaluation

4.1 Experimental Setup

We implemented a dedupe storage simulator based on the DiskSim simulator. DiskSim is widely adopted for disk-based storage subsystem simulations because it is very accurate and highly configurable[33]. Specifically, the disk drive module simulates modern disk drives in extreme detail and has been carefully validated against several production disks[33].

The underlying storage includes 9 individual disks, each providing 17 916 240 blocks (8.54 GB) of storage capacity. It is configured as RAID0 to improve performance and enough storage space to accommodate all the chunks in the backup datasets. The stripe unit size is set to 32 KB. The chunk indexing (hash index table) is based on Google Sparsehash. For read and write performance measurements, it ignores the elapsed time to execute the typical dedupe or the proposed scheme because it is much smaller than storage I/O time. For the dedupe, it uses a fix-sized 2 MB container and each container read (write) accesses 64 stripe units, where each individual disk serves about 7∼8 stripe unit reads (writes).

Six backup datasets, traced in real data backup environments, are employed①. Table 2 summarizes the dataset characteristics. Each dataset consists of 2 or 5 concatenated, weekly full-backup data streams. The ds-1, ds-2, and ds-3 datasets were obtained from all exchange server data encoded in different ways. The ds-4 contains system data for a revision control system. The ds-5 includes the data from the */var* directory in the same machine. The ds-6 contains the data from home directories in a workstation supporting several users. For experimental purposes, all datasets except the last (ds-6) were truncated to 20 GB in order for them to have about 1 900k total chunks. Each dataset contains chunked-data streams by using variable-length chunking with an average chunk size of 8 KB.

Table 2. Characteristics of the Six Real Backup Datasets

Dataset	Number of Chunks	DGR	Size (GB)	Number of Ver.
ds-1	1 970 836	0.29	20	5
ds-2	1 960 555	0.37	20	5
ds-3	1 951 271	0.97	20	5
ds-4	1 892 285	0.50	20	5
ds-5	1 922 420	0.20	20	5
ds-6	771 465	0.38	8	2

①For convenience, we shorten each trace name (i.e., ds-1∼ds-6). Original trace names are exchange (ds-1), exchange_is (ds-2), exchange_mb (ds-3), fredp4 (ds-4), fredvar (ds-5), and sazzala (ds-6).

Each data stream in the datasets consists of a sequence of chunk records each of which specifies the key chunk information including 20-byte chunk ID (SHA-1 hash value), its LBA information, dataset and version IDs (Ver.), and a chunk size (not compressed size). The deduplication gain ratio (DGR) represents the ratio of the stored data stream size to the original data stream size. Most datasets except ds-3 contain many duplicated chunks. The proposed design is compared with the LRU approach which is a widely adopted caching scheme in dedupe storage systems[16,34-35].

4.2 Experimental Results

Fig.8 shows the LRU and base lookahead design read performance with various cache sizes. Note that the cache size of 2, 4, and 8 means the number of 2 MB units in a data container. Thus, the cache size of 2, 4, and 8 corresponds to 4 MB, 8 MB, and 16 MB cache respectively.

Both Figs.8(a) and 8(b) exhibit very typical read performance for dedupe backup datasets. For each backup version, its successive version is likely to contain more duplicate data chunks (please refer to Figs.3(a) and 3(b)), which leads to read performance degradation. Therefore, performance generally continues degrading with each backup version. As plotted, even when only using a cache replacement algorithm, the base lookahead design exhibits better read performance than LRU for all cache sizes because it exploits future access information. The proposed scheme improves dedupe read performance by an average of 14.5% and 16.8% respectively. For the Ver-1 in both Fig.8(a) and Fig.8(b), all of the schemes exhibit the same performance because all the chunks are unique (99.9% in ds-1 and 100% in ds-2). Therefore, no cache algorithm can help improve the performance. On the other hand, the other four datasets (ds-3∼ds-6) show difference performance in the first backup version (Ver-1) because they have duplicate chunks, even in the Ver-1. You can refer these to the dataset analysis in Fig.3 and Table 1.

Unlike the other five datasets, ds-4 shows a somewhat different performance pattern for each backup version. Its performance fluctuates from version to version. This is also closely related to the amount of duplicate data chunks and its data container access pattern. As shown in Fig.3(d), Ver-1, Ver-3 and Ver-5 do not have so many shared chunks (i.e., vertical lines) as both Ver-2 and Ver-4. Consequently, Ver-1, Ver-3 and Ver-5 exhibit higher read performance than the others. Now, Fig.9∼Fig.12 depict the performance improvement with the extended cache design with various configurations.

First of all, Fig.9 explores the impact of both a hot

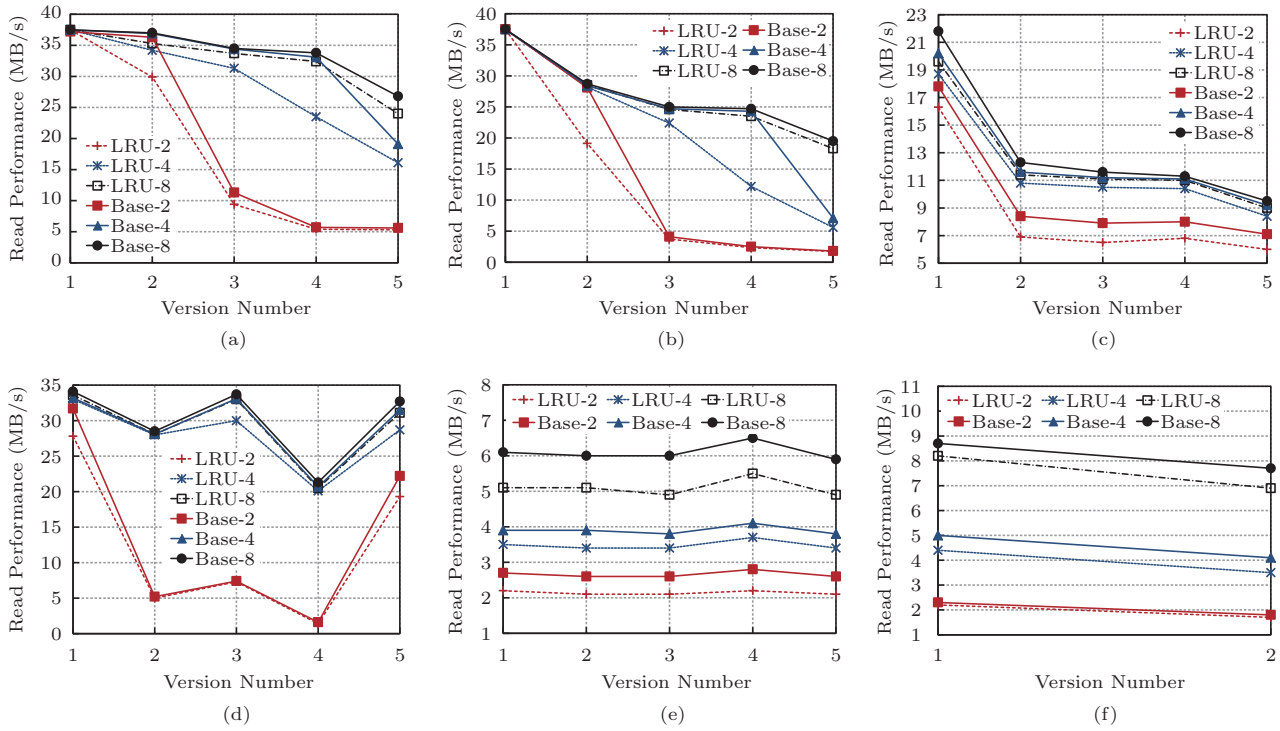


Fig.8. Base lookahead cache vs LRU. (a) ds-1. (b) ds-2. (c) ds-3. (d) ds-4. (e) ds-5. (f) ds-6.

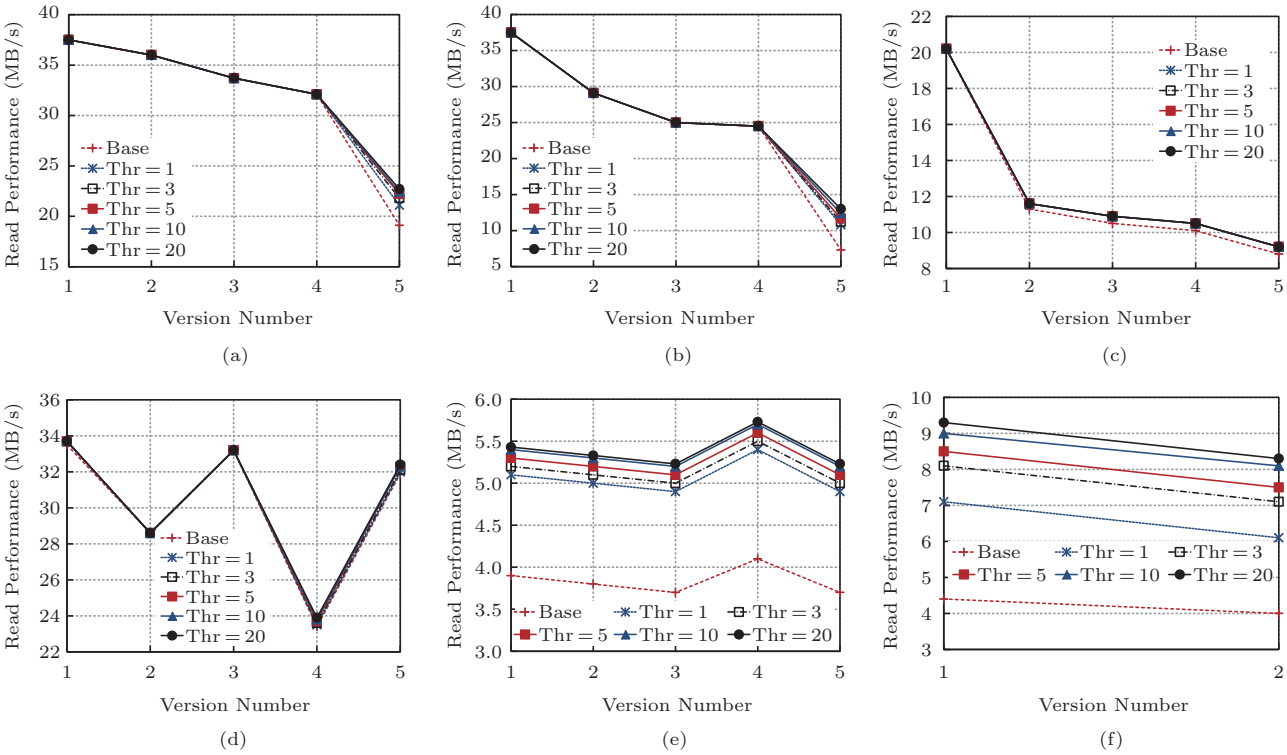


Fig.9. Base vs extended design with diverse hot thresholds. (a) ds-1. (b) ds-2. (c) ds-3. (d) ds-4. (e) ds-5. (f) ds-6.

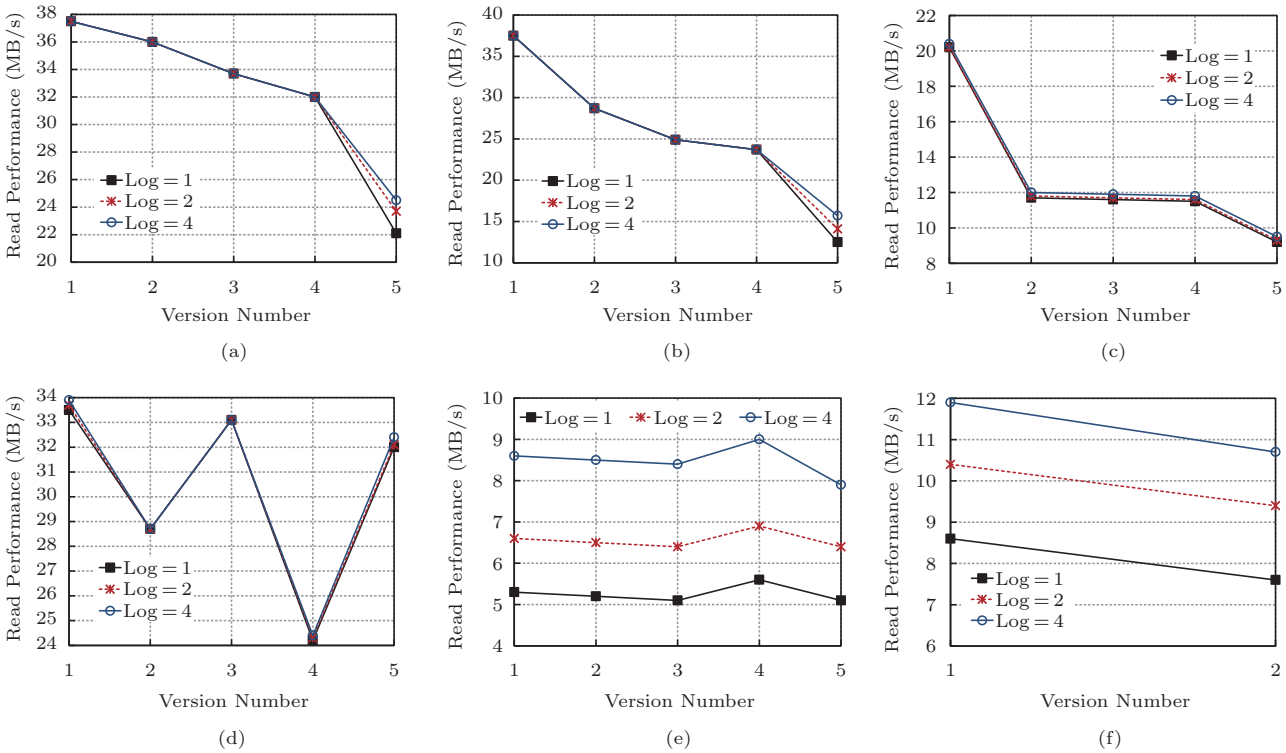


Fig.10. Impact of log buffer size. Here, a log size corresponds to the number of a container (e.g., log = 2 stands for 2 × the size of a container). (a) ds-1. (b) ds-2. (c) ds-3. (d) ds-4. (e) ds-5. (f) ds-6.

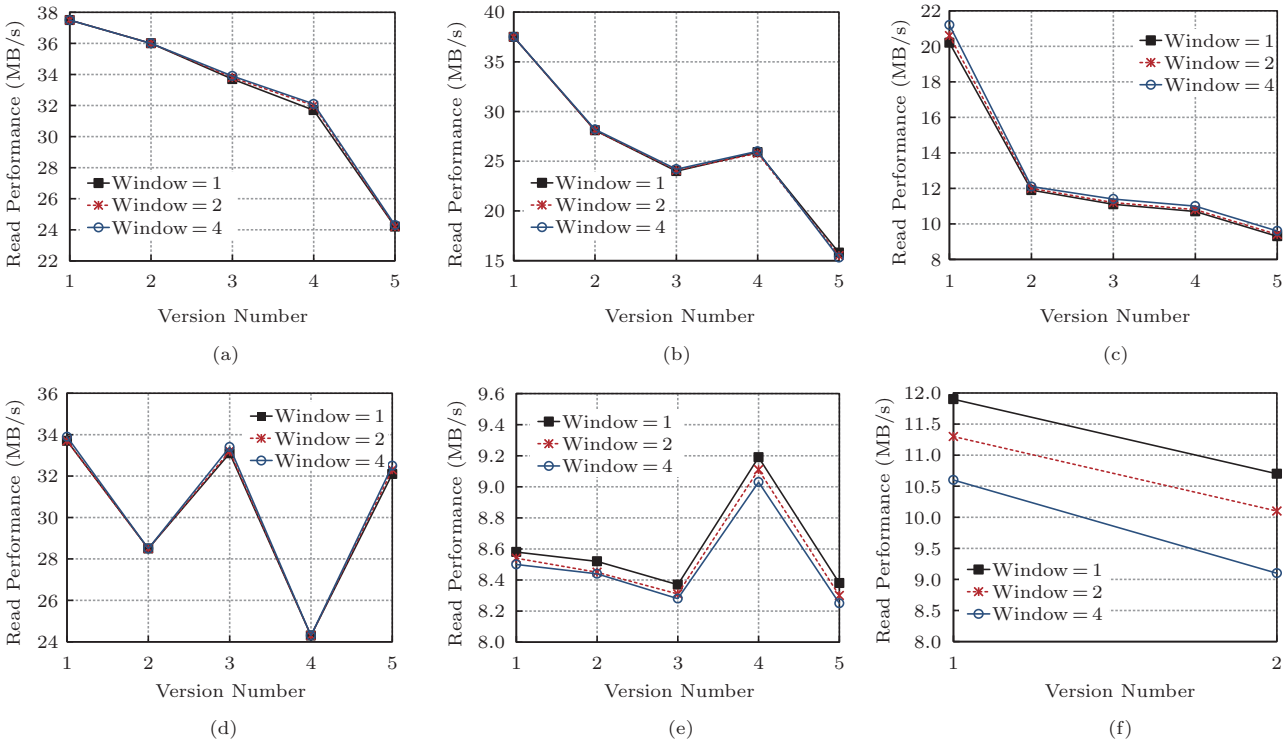


Fig.11. Impact of a window size (window). Here, the unit of a future window corresponds to the number of a data container. Thus, it can look ahead the following number of chunks: the window unit × the number of data chunks in one container. (a) ds-1. (b) ds-2. (c) ds-3. (d) ds-4. (e) ds-5. (f) ds-6.

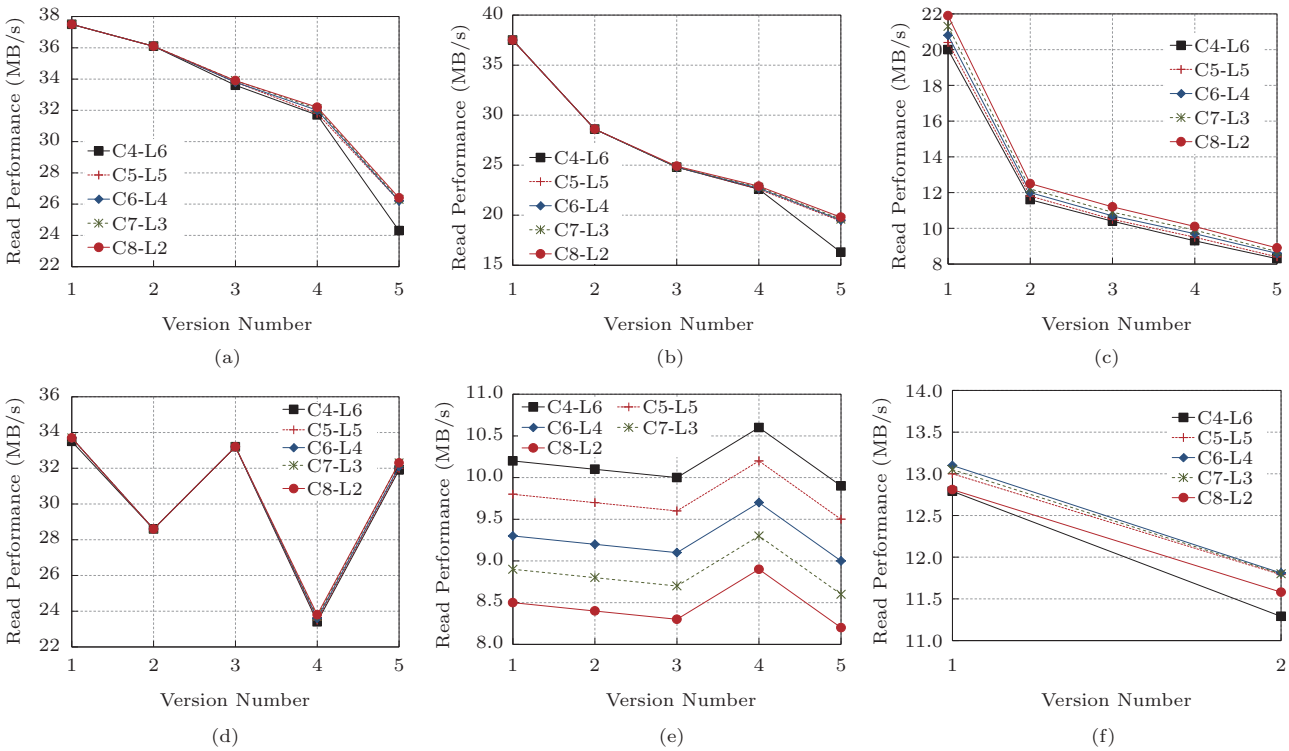


Fig.12. Impact of diverse cache and log buffer sizes. Here, a total space (cache plus log buffer) is fixed. C and L stand for the cache size and the log buffer size respectively. (a) ds-1. (b) ds-2. (c) ds-3. (d) ds-4. (e) ds-5. (f) ds-6.

threshold value (Thr) and the extended design's performance improvement. Hot data identification plays an important role in making decisions about logging future container data chunk accesses into the log buffer. The extended scheme stores the remaining future access chunks in the small buffer only if the container is identified as cold. Here, the hot threshold value stands for the percentage of accessed data chunks in a container. It can be intuitively expected that the higher a hot threshold, the better the read performance. This is because the buffer will be able to store more future access chunks. However, the proposed design with a threshold higher than 3% or 5% does not lead to higher performance gain. In addition, it is observed that the extended design (with a log buffer) does not considerably improve the performance in ds-1, ds-2, ds-3, and ds-4 (2.3% improvement on average). However, in both ds-5 and ds-6, it significantly improves the read performance by an average of 63.1%. These results stem from the fact that there are many duplicates even in Ver-1. Thus, many containers can be classified as cold. Consequently, a large number of future reference chunks can be stored in the log buffer.

Fig.10 exhibits the impact of a log buffer size. It is observed that it has a more impact on both ds-5 and ds-6 than ds-1∼ds-4 (47.8% vs 9.8% on average). Fig.10 also supports the experimental results shown in Fig.9. Both Fig.9 and Fig.10 show very similar performance patterns, because the performance gains of the extended design fundamentally originate from the log buffer. Moreover, the extended design outperforms LRU by an average of 64.3% in ds-5 and ds-6.

The sliding window size is another factor to be discussed since it implies how much a lookahead scheme can look ahead for future references (Fig.11). Interestingly, the impact of a bigger window is almost negligible in ds-1∼ds-4 (less than 0.5%) and it even degrades the performance in ds-5 and ds-6 (6.6% on average). This is because farther future accesses in a big window can contaminate a cache space, which can lead to low cache utilization in the cache policies.

Lastly, Fig.12 explores the impacts of various cache and log buffer sizes. It first fixes the total cache space and then varies both cache sizes and log buffer sizes. As in Fig.10, it is observed that a read cache size is a more important factor in ds-1∼ds-4. Thus, it is suggested that more space be assigned to a read cache (instead of the log buffer) for those types of datasets. On the other hand, a larger log buffer has a more impact on the read performance in both ds-5 and ds-6. Because both ds-5 and ds-6 contain many duplicate chunks (even in Ver-1), the big read cache does not have so much impact as a log buffer. Cache utilization will not be relatively high due to many shared chunks. Therefore, assigning more space to the log buffer for these types of datasets will help improve dedupe read performance.

5 Related Work

Zhu *et al.* in [8] put an emphasis on the importance of dedupe storage read performance, for data recovery in particular. They also addressed that read performance substantially decreased during a dedupe process. I/O deduplication[36] is a selective duplication scheme to increase read/write performance by reducing disk head movement. iDedupe[1] is a primary inline dedupe system taking the read performance issue into consideration. It tries to exploit both spatial locality by selectively de-duplicating primary data and temporal locality by maintaining dedupe metadata completely in memory, not on disk. Nam *et al.*[19] introduced an indicator for the degraded read performance named chunk fragmentation level (CFL) and observed a strong correlation between the CFL value and the read performance under backup datasets.

Nam *et al.*[15] proposed a novel backup deduplication storage system in order to assure required read performance. It adopts a selective deduplication approach based on a read performance monitoring factor (CFL). Lillibridge *et al.*[16] proposed three mechanisms (a speed factor, container capping, and forward assembly) to improve backup system restore speeds. This work also employs a selective deduplication based on their performance monitoring factor (i.e., a speed factor). HAR[34] uses historical backup system information to identify and rewrite fragmented chunks so that it can remove the merging operation in containers. In addition, it tries to reduce garbage collection overhead.

Mao *et al.*[29] proposed the SAR (SSD-assisted read) scheme to accelerate read performance for deduplication-based storage systems. SAR stores unique data chunks with a high reference count, small size, and non-sequentiality into SSDs to utilize the high random-read performance of SSDs. Li *et al.*[37] proposed RevDedup to improve not only restore performance, but also backup performance and backup deletion performance. RevDedup first applies coarse-grained (i.e., large size unit) inline deduplication, and then applies fine-grained (small size units) out-of-line deduplication to improve storage efficiency. Tan *et*

al.[38] proposed De-Frag to improve deduplication performance by reducing data chunk fragmentation (i.e., de-linearization). De-Frag also selectively duplicates data chunks in a deduplication system by monitoring the spatial locality of each chunk group and adding redundant chunks. Fu *et al.* in [3] provided an efficient backup system design guide via extensive studies on the fundamental tradeoffs of existing modern backup systems. Unlike these studies, this paper focuses specifically on a dedupe storage system read cache design.

6 Conclusions

This paper addressed a data deduplication (dedupe) read performance problem and proposed a novel dedupe read cache design, named lookahead read cache, for backup applications. The key idea is inspired by the following special backup application workload characteristic: read sequences are identical to write sequences. Based on this consideration, the proposed design can exploit future read access patterns by evicting a cache victim with a smallest future reference count. Moreover, unlike the existing selective duplication approach, it does not sacrifice write performance for read performance improvements since a new read cache design in dedupe storage systems does not affect a write buffer design. The base lookahead read cache design was extended with a small log buffer and a hot data identification scheme. It stores a small number of future access chunks in the log buffer before the cold container is evicted.

The experiments with real-world workloads demonstrated that the lookahead read cache shows a better performance than the existing LRU cache design.

Acknowledgment We would like to thank Dr. Guanlin Lu (EMC Corporation, USA) for his valuable comments. We also would like to thank David Schwaderer (Samsung Semiconductor Inc., USA) for his comments and proofreading.

References

[1] Srinivasan K, Bisson T, Goodson G, Voruganti K. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. the 10th USENIX Conference on File and Storage Technologies*, February 2012, pp.299-312.

[2] Nam Y, Park D, Du D H C. Virtual USB drive: A key component for smart home storage architecture. In *Proc. IEEE International Conference on Consumer Electronics*, January 2012, pp.23-24.

[3] Fu M, Feng D, Hua Y, He X B, Chen Z N, Xia W, Zhang Y C, Tan Y J. Design tradeoffs for data deduplication performance in backup workloads. In *Proc. the 13th USENIX Conference on File and Storage Technologies*, February 2015, pp.331-344.

[4] Fu Y J, Xiao N, Liao X K, Liu F. Application-aware client-side data reduction and encryption of personal data in cloud backup services. *J. Comput. Sci. Technol.*, 2013, 28(6): 1012-1024.

[5] Dong W, Douglis F, Li K, Pattersonet H, Reddy S, Shilane P. Tradeoffs in scalable data routing for deduplication clusters. In *Proc. the 9th USENIX Conference on File and Storage Technologies*, February 2011, pp.15-29.

[6] Dubnicki C, Gryz L, Heldt L, Kaczmarczyk M, Kilian W, Strzelczak P, Szczepkowski J, Ungureanu C, Welnicki M. HYDRAstor: A scalable secondary storage. In *Proc. the 7th Conference on File and Storage Technologies*, February 2009, pp.197-210.

[7] Debnath B, Sengupta S, Li J. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. USENIX Conference on USENIX Annual Technical Conference*, June 2010.

[8] Zhu B, Li K, Patterson H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. the 6th USENIX Conference on File and Storage Technologies*, February 2008, Article No. 18.

[9] Meyer D T, Bolosky W J. A study of practical deduplication. In *Proc. the 9th USENIX Conference on File and Storage Technologies*, February 2011.

[10] Seo M K, Lim S H. Deduplication flash file system with PRAM for non-linear editing. *IEEE Transactions on Consumer Electronics*, 2010, 56(3): 1502-1510.

[11] Min J, Yoon D, Won Y. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 2011, 60(6): 824-840.

[12] Fineberg S, Rabinovici-Cohen S. Long term information retention. In *Proc. Storage Developer Conference*, September 2011. http://www.snia.org/events/storage-developer2011/agenda2011/abstracts#long_term_ret, Dec. 2016.

[13] Kay D, Maybee M. Aspects of deduplication. Storage Networking Industry Association (SNIA) Tutorial, January 2010. http://www.snia.org/sites/default/education/tutorials/2010/spring/file/MarkMaybee-DominicKay_Aspects_of_Deduplication.pdf, Dec. 2016.

[14] Kim Y. Consolidate more: High-performance primary deduplication in the age of abundant capacity. Hitachi Data Systems, March 2013. http://www.slideshare.net/hdscorp/consolidate-more-high-performance-primary-deduplication-in-the-age-of-abundant-capacity, Dec. 2016.

[15] Nam Y, Park D, Du D H C. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proc. the 20th IEEE International Symposium on Modeling, Analysis and Simulations of Computer and Telecommunication Systems*, August 2012, pp.201-208.

[16] Lillibridge M, Eshghi K, Bhagwat D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. the 11th USENIX Conference on File and Storage Technologies*, February 2013, pp.183-198.

[17] Welnicki M, Szczepkowski J, Dubnicki C. System and method for deduplication of distributed data. US Patent 9, 256, 368, February 2016. http://patft.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=9256368.PN.&OS=PN/9256368&RS=PN/9256368, Dec. 2016.

[18] Efstathopoulos P, Guo F L. Rethinking deduplication scalability. In *Proc. the 2nd USENIX Conference on Hot Topics in Storage and File Systems*, June 2010.

[19] Nam Y, Lu G L, Park N, Xiao W J, Du D H C. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proc. the 13th IEEE International Conference on High Performance Computing and Communications*, September 2011, pp.581-586.

[20] Rabin M O. Fingerprinting by random polynomials. Technical Report TR-1581, 1981. http://www.citeulike.org/user/dmeister/article/2706665, Nov. 2016.

[21] Liu C Y, Gu Y, Sun L C, Yan B, Wang D S. RADMAD: High reliability provision for large-scale de-duplication archival storage systems. In *Proc. the 23rd International Conference on Supercomputing*, June 2009, pp.370-379.

[22] Jin K R, Miller E L. The effectiveness of deduplication on virtual machine disk images. In *Proc. SYSTOR 2009: The Israeli Experimental Systems Conference*, May 2009, Article No. 7.

[23] Lim H, Fan B, Andersen D G, Kaminsky M. SILT: A memory-efficient, high-performance key-value store. In *Proc. the 23rd ACM Symposium on Operating Systems Principles*, October 2011, pp.1-13.

[24] Park N, Lilja D J. Characterizing datasets for data deduplication in backup applications. In *Proc. IEEE International Symposium on Workload Characterization*, December 2010.

[25] Bhagwat D, Eshghi K, Long D D E, Lillibridge M. Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. the 17th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, September 2009.

[26] Lillibridge M, Eshghi K, Bhagwat D, Deolalikar V, Trezise G, Camble P. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *Proc. the 7th Conference on File and Storage Technologies*, February 2009, pp.111-123.

[27] Lu G L, Jin Y, Du D H C. Frequency based chunking for data de-duplication. In *Proc. IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, August 2010, pp.287-296.

[28] Tsuchiya Y, Watanabe T. DBLK: Deduplication for primary block storage. In *Proc. the 27th IEEE Symposium on Massive Storage Systems and Technologies*, May 2011.

[29] Mao B, Jiang H, Wu S Z, Fu Y J, Tian L. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Transactions on Storage*, 2014, 10(2): Article No. 6.

[30] Wallace G, Douglis F, Qian H W, Shilane P, Smaldone S, Chamness M, Hsu W. Characteristics of backup workloads in production systems. In *Proc. the 10th USENIX Conference on File and Storage Technologies*, February 2012.

[31] Park D, Du D H C. Hot data identification for flash-based storage systems using multiple bloom filters. In *Proc. the 27th IEEE Symposium on Mass Storage Systems and Technologies*, May 2011.

[32] Megiddo N, Modha D. ARC: A self-tuning, low overhead replacement cache. In *Proc. the 2nd USENIX Conference on File and Storage Technologies*, March 2003, pp.115-130.

[33] Bucy J S, Schindler J, Schlosser S W, Ganger G R. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, 2008. http://www.pdl.cmu.edu/PDL-FTP/DriveChar/CMU-PDL-08-101.pdf, Nov. 2016.

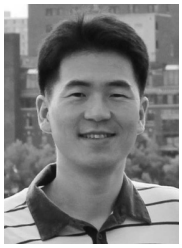
[34] Fu M, Feng D, Hua Y, He X B, Chen Z N, Xia W, Huang F T, Liu Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proc. USENIX Conference on USENIX Annual Technical Conference*, June 2014, pp.181-192.

[35] Li W J, Jean-Baptise G, Riveros J, Narasimhan G, Zhang T, Zhao M. CacheDedup: In-line deduplication for flash caching. In *Proc. the 14th USENIX Conference on File and Storage Technologies*, February 2016, pp.301-314.

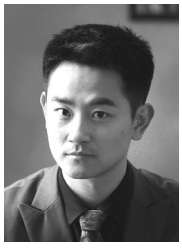
[36] Koller R, Rangaswami R. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proc. the 8th USENIX Conference on File and Storage Technologies*, February 2010.

[37] Li Y K, Xu M, Ng C H, Lee P P C. Efficient hybrid inline and out-of-line deduplication for backup storage. *ACM Transactions on Storage*, 2015, 11(1): Article No. 2.

[38] Tan Y J, Yan Z C, Feng D, He X B, Zou Q, Yang L. De-Frag: An efficient scheme to improve deduplication performance via reducing data placement de-linearization. *Cluster Computing*, 2015, 18(1): 79-92.



Dongchul Park is currently a research scientist in Memory Solutions Laboratory (MSL) at Samsung Semiconductor Inc. in San Jose, California. He received his Ph.D. degree in computer science and engineering at the University of Minnesota–Twin Cities, Minneapolis, in 2012, and was a member of Center for Research in Intelligent Storage (CRIS) group under the advice of Professor David H. C. Du. His research interests focus on storage system design and applications including non-volatile memories, in-storage computing, big data processing, Hadoop MapReduce, data deduplication, key-value store, cloud computing, and shingled magnetic recording (SMR) technology.



Ziqi Fan is a Ph.D. candidate in computer science and Engineering at the University of Minnesota–Twin Cities, Minneapolis. He is a member of Center for Research in Intelligent Storage (CRIS) group under the advice of Professor David H. C. Du. He received his B.E. degree in network engineering in the School of Software at Dalian University of Technology, Dalian, in 2012. His research interests focus on buffer cache policies, SSD and non-volatile memories, file systems and cloud storage, and data deduplication.

Young Jin Nam is currently a principal software engineer at Oracle Corporation in Santa Clara, California, USA. He was a visiting professor at the University of Minnesota–Twin Cities in 2011 and was an associate professor in the School of Computer and Information Technology at Daegu University in South Korea from 2004 to 2012. He received his Ph.D. degree in computer science and engineering at Pohang University of Science and Technology (POSTECH), South Korea, in 2004. His research interests cover data deduplication, key-value store, high-performance storage architecture, and new storage technologies such as flash and phase change memory (PCM).

David H. C. Du is currently the Qwest Chair Professor in computer science and engineering at the University of Minnesota–Twin Cities, Minneapolis, and the center director of the Center of Research in Intelligent Storage (CRIS). He received his Ph.D. degree in computer science from University of Washington, Seattle, in 1981, and his current research focuses on intelligent storage systems, multimedia computing, sensor networks, and cyber physical systems. He was a Program Director (IPA) at National Science Foundation (NSF) CISE/CNS Division from March 2006 to August 2008, and has served as a chair, program committee chair, and general chair for many conferences.