Improving the Restore Performance via Physical -Locality Middleware for Backup Systems

Pengfei Li, Yu Hua, Qin Cao, Mingxuan Zhang
Wuhan National Laboratory for Optoelectronics, School of Computer
Huazhong University of Science and Technology
Corresponding Author: Yu Hua (csyhua@hust.edu.cn)

Abstract

Data deduplication as an important middleware plays an essential role in current backup systems due to the high space efficiency, which however suffers from low restore performance, since the chunks are heavily fragmented. Existing systems leverage caching and rewriting schemes to reduce the number of container reads to restore the original data. However, when storing a large number of versions, the caching schemes become inefficient since each cached container consists of a few required chunks due to the exacerbated fragmentation, while the rewriting schemes consume more space to store the duplicate chunks.

In this paper, we explore and exploit the behaviors of the fragmented chunks in the backup systems. Based on the observations, we propose the open-sourced HiDeStore[1], a High-performance Deduplication and reStore backup system without decreasing the deduplication ratio. The main insight is to enhance the physical locality for the new backup versions during the deduplication process. Thus, the chunks are stored closely and the number of container reads significantly decreases to restore the original data. Compared with state-of-the-art deduplication and restore schemes, HiDeStore reduces the index lookup overhead by 38% and improves the restore performance by up to 1.6×, without decreasing the deduplication ratio.

CCS Concepts: • **Software and its engineering** → Software system structures.

Keywords: deduplication, backup systems, physical-locality

ACM Reference Format:
Pengfei Li, Yu Hua, Qin Cao, Mingxuan Zhang. 2020. Improving the Restore Performance via Physical -Locality Middleware for Backup Systems. In *21st International Middleware Conference (Middleware '20), December 7–11, 2020, Delft, Netherlands*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3423211.3425691

1 Introduction

The data grow exponentially in the widely used applications, such as IoT embeddings, artificial intelligence and cloud computing, which require efficient and large-scale storage capacities [7, 17, 18]. To save space and improve storage efficiency, data deduplication [31, 41] becomes an efficient middleware to eliminate the duplicate data, and has been widely used in current storage systems [11, 24–26, 32, 39], especially for storage backup systems [14, 19, 30].

In the chunk-based deduplication systems [21, 31, 41], the data streams are divided into chunks with on average 4-8KB, which are further represented by fingerprints calculated via a cryptographic hash function, such as SHA-1 and MD5. Since the probability of a hash collision is much smaller than that of a hardware error [31], the chunks that share the same fingerprint are identified as duplicate chunks. To save the space, the duplicate chunks are stored in the *containers* (i.e., typical 4MB space to store the chunks) only once on the persistent storage, such as HDD or SSD. The data streams are then represented as the lists of the chunk references (i.e., refer to the physical locations of the chunks), called *recipes*, which are used to restore the original data.

However, the deduplication process incurs severe fragmentation problem [13, 16, 20], which hinders the restore performance. Specifically, in a continuous data stream, the unique chunks are stored into new containers when arriving, and the identified duplicate chunks are referenced to the containers that have been written for a long time. As a result, we have to read a large number of containers to obtain all the chunks, incurring lots of expensive I/Os to the persistent storage. Moreover, the

[1] The source code of HiDeStore is available at https://github.com/iotlpf/HiDeStore

Middleware '20, December 7–11, 2020, Delft, Netherlands
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-8153-6/20/12...$15.00
https://doi.org/10.1145/3423211.3425691

chunk fragmentation problem is exacerbated over time, which severely decreases the restore performance.

Existing systems leverage two kinks of schemes to improve the restore performance. First, the caching-based schemes (including container-based caching [13, 16, 28], chunk-based caching [9, 20, 22] and forward assembly [9, 20]) are proposed to reduce the number of container reads. The main insight is to exploit the logical locality of the backup stream, i.e., the chunks are stored in the same order as they first appear in the stream. Therefore, the read containers have high probability to contain the subsequent chunks of the same data stream, which exhibits good cache-friendly locality. Second, some schemes rewrite the duplicate chunks to enhance the physical locality (i.e., the physical layout of a backup stream after deduplication) of the data stream [8, 13, 16, 27]. By writing some duplicate chunks to the new containers, the chunks belonging to the same data streams are stored closely, hence fewer containers are read to restore the original data.

However, existing schemes become inefficient when storing a large number of backup versions due to the following limitations. First, the caching schemes fail to alleviate the fragmentation problem, since the chunks are stored into more containers as the backup data increase. As a result, one container contains a few chunks belonging to the same data stream, which becomes inefficient to be cached. Second, although rewriting some duplicate chunks enhances the physical locality, the deduplication ratio decreases due to the existence of duplicate chunks. We have to allocate extra space to store the rewritten chunks, which is exacerbated over time. For example, even if the deduplication ratio decreases 1%, 10GB extra space is required for 1TB unique data, which is used to store the rewritten data.

To overcome the limitations of existing schemes, we explore and exploit the behaviors of the fragmented chunks in the backup systems. We design a heuristic experiment to figure out how the chunks are stored during the deduplication process. We have two important observations from our experimental results. First, the high redundancy arises between adjacent backup versions, and the chunks that don't appear in current backup version have low probability to appear in the subsequent backup versions. Second, a new backup version is more likely to be restored than the old ones, which means that the high restore performance for the new backup version is more important than the old ones.

Based on the observations, the fragmented chunks can be eliminated by modifying the deduplication process. We propose a *scalable deduplication scheme* with high deduplication/restore performance and deduplication ratios, called HiDeStore. The ***scalability*** in this paper is interpreted that the proposed scheme provides high restore performance over time, which is efficient even when a large number of backup versions are stored. The main insight is to identify the chunks that are more likely to be shared by the subsequent backup versions, called *hot chunks*, and store these chunks together to enhance the physical locality for the new backup versions. The other chunks (i.e., have the low probability to be shared by the new backup versions) are stored in the containers as the traditional deduplication schemes, called *cold chunks*.

Specifically, the whole deduplication process consists of three steps. (1) Classify the hot and cold chunks. We use two hash tables in the fingerprint cache to respectively store the chunks of the last and current backup versions. Each hash table contains the fingerprints and corresponding location information of the chunks. (2) Filter and store the contents of the hot and cold chunks. The contents of the coming unique chunks are temporarily stored in the active containers when deduplicating one backup version, and the contents of the identified cold chunks are stored in the archival containers after processing the backup versions. (3) Update the recipes for restoring the original data in the future. We construct a recipe link list to reduce the updating overhead, and further optimize the process of recipe searching by periodically eliminating the dependency among recipes. Compared with state-of-the-art deduplication and restore schemes, HiDeStore reduces the index lookup overhead by 38% and improves the restore performance by up to 1.6×. Moreover, HiDeStore keeps high deduplication ratio while incurring acceptable overheads. Our proposed scheme is efficient for archival backup systems, e.g., backup all versions of the software and the system snapshots for users.

In this paper, we have the following contributions.

- ***High deduplication performance without the decrease of deduplication ratio.*** We explore the workload characteristics in backup systems and only cache the chunks with high probability to be deduplicated, without frequently accessing the disks. Therefore, the deduplication throughput is improved. The duplicate chunks have high probability to be hit in the fingerprint cache, thus achieving the high deduplication ratio.
- ***High restore performance for new backup versions.*** Based on the chunk distribution among different backup versions, we filter and store the cold and hot chunks in different containers, thus enhancing the physical locality for new backup versions. As a result, we achieve high restore performance for new backup versions due to fewer container reads than existing schemes.
- ***Low Overheads.*** Unlike the traditional schemes, HiDeStore doesn't need extra space to store the duplicate chunks and the full index table due to the high

deduplication ratio. Moreover, removing expired backup version in HiDeStore is easy to implement without the need of garbage collection.

The remainder of this paper is organized as follows. Section 2 introduces the background and motivations. Section 3 shows our observations on the backup systems. Section 4 illustrates the design of HiDeStore. In Section 5, we compare our design with the state-of-the-art schemes and show the experimental results. Section 6 describes the related work. Finally, we summarize our work in Section 7.

2 Background

2.1 The Deduplication Process

The chunk-based deduplication is an efficient approach to improve the space utilization, which has been widely used in current storage systems [11, 24–26, 32, 39]. A deduplication system usually consists of two phases, including deduplication and restore phases, as shown in Figure 1. In this paper, we focus on the in-line deduplication [9, 10, 13, 16, 21, 23, 28, 41], i.e., the data are deduplicated once they are generated.

In the deduplication phase, ❶the coming data stream is divided into multiple chunks (e.g., on average 4-8KB [41]) via various chunking algorithms, such as TTTD chunking [12], Rabin-based CDC [26], FastCDC [37]. ❷The obtained chunks are represented as 20-byte fingerprints that are calculated through a secure hash function, e.g., SHA-1 [31]. To eliminate the duplicate chunks, we only need to compare the 20-byte fingerprints, rather than the original 4KB data. Because the probability of a hash collision is much smaller than that of a hardware error [31]. ❸The chunks with the same fingerprint are identified to be duplicated, and otherwise the chunks are unique. However, as the data increase, the generated fingerprints overflow the limited memory, which become a system bottleneck to search the fingerprints [21, 41]. Existing deduplication systems maintain an efficient fingerprint cache in the memory to accelerate the deduplicate process by exploiting the locality and similarity [6, 21, 35, 41]. ❹When the coming fingerprints miss in the cache, the fingerprints are further searched in the full index table on disks to obtain high deduplication ratio. However, searching the full index table is optional, depending on the deduplication algorithms and workload characteristics. ❺Finally, the unique chunks are stored into typical 4MB containers as the chunks arrive. The metadata (e.g., fingerprint, chunk size and container ID) of all the chunks are recorded in a *recipe* [41] for the data recovery. Specifically, the data structure of the recipe is a chunk list, and each item contains a fingerprint (20-byte) (i.e., corresponding to a chunk), the ID of the container (4-byte) containing the chunk, and the offset (4-byte) in the container. The size of each item in the recipe is 28-byte, and the total size of the recipe is determined by the number of the chunks.

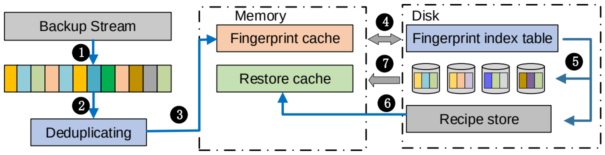


Figure 1. The deduplication and restore phases.

The data need to be restored from system crashes or version rollbacks [16, 20], which consists of the following steps. ❻To restore the original data, the recipe is accessed to determine the chunks need to be read and the locations to obtain the corresponding chunks. ❼According to the instructions of the recipe, we assemble the data stream in memory in the chunk-by-chunk manner, which requires frequent disk accesses. To improve the restore performance, the caching schemes are used to reduce the number of container reads. However, the restore performance is affected by the physical locality of the chunks, since the frequent and random disk accesses cause significantly high overheads.

2.2 Chunk-based Access Bottleneck

In large-scale deduplication systems [21, 41], searching the fingerprint indexes to identify the duplicate and unique chunks is usually a performance bottleneck. Since the number of fingerprints increases proportionally with the stored data, the index table possibly overflows the limited memory, hence involving frequent disk accesses for index lookups. The fingerprint index table is actually a key-value store, where the key is a fingerprint and the value points to the chunk. Each key consumes 20 bytes (e.g., calculated by SHA-1) to represent a typical 4KB chunk, and thus indexing 4TB unique chunks requires at least 20GB to store the keys, which consumes a large amount of memory space. More importantly, the number of keys increases as new data arrive, which is difficult to put the whole index table in the memory.

To avoid the frequent accesses to the full index table on disk, existing deduplication systems store partial indexes in memory (i.e., fingerprint cache) to improve the deduplication performance. As shown in Figure 1, the chunks hitting the fingerprint cache are considered as duplicated chunks, which are not stored for space saving. Other chunks (i.e., miss in the fingerprint cache) are searched in the index table on disk to further identify the duplicate and unique chunks. Only the unique chunks are stored. During this process, the fingerprint cache with the high hit ratio will decrease the access to disks and improve the deduplication throughput. Thus various

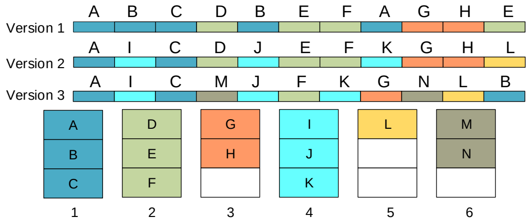


Figure 2. The chunk fragmentation problem. *The order of versions is determined by the generation time.*

schemes aiming to improve the hit ratio of the fingerprint cache are proposed [21, 41].

Some schemes [10, 21, 23, 41] make full use of the locality characteristic, i.e., the chunks among different backup streams appear in approximately the same order with a high probability. The chunks following the searched chunks are prefetched into the fingerprint cache during one disk access, which significantly improves the hit ratio. Moreover, only partial indexes are stored according to the sampling approaches to reduce the memory consumption [21, 38]. Nevertheless, for the workloads that have little or no locality, the locality-based approaches produce poor performance. In these cases, the similarity-based approaches are proposed [6, 35]. The two data streams are highly similar if they share many chunks. However, the similarity-based approaches decrease the deduplication ratios, depending on the similarity among data streams. Because only the most similar data streams are searched to improve deduplication throughput, which overlooks the full index table.

Existing schemes involve different amounts of disk accesses, depending on the locality and similarity of the workloads. Moreover, we have to make a trade-off between deduplication ratio and throughput, since the high deduplication throughput is achieved by avoiding the search for the full index table, depending on the workload characteristics.

2.3 Chunk Fragmentation Problem

Unlike the deduplication phase that searches the metadata (i.e., fingerprint indexes), the restore phase accesses the real data of the chunks according to the recipe. The restore performance suffers from the chunk fragmentation problem [8, 9, 13, 16, 20, 30], i.e., the chunks of the same data stream are scattered into various containers, causing frequent disk accesses during the recoveries. The chunk fragmentation problem comes during the deduplication phase. The unique chunks are stored in new containers, while the identified duplicate chunks refer to the containers that have been stored for a long time. Thus, the chunks are separately stored and the physical locality is destroyed.

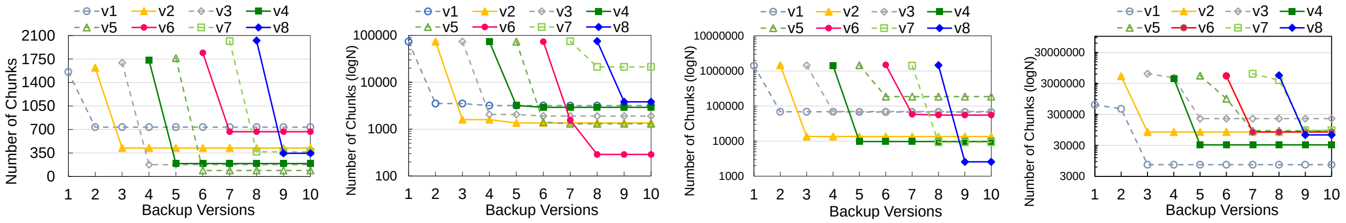
Figure 2 illustrates how the chunk fragmentation problem arises with the assumption that each container contains at most 3 chunks. During the deduplication phase, the unique chunks are stored in the containers when they arrive. The chunks belonging to the first data stream are stored in Containers 1, 2 and 3. For the second data stream, the identified duplicate chunks (e.g., Chunks A, C, D, E, F, G, H) are not stored, while the unique chunks (e.g., Chunks I, J, K, L) are stored in Containers 4 and 5. As a result, we need to access 5 containers to restore the second backup stream. The same storing mechanism is applied to the third data stream, which needs to access 6 different containers to restore the original data. Moreover, such chunk fragmentation problem is exacerbated over time.

Various chunk-based and container-based caching schemes [9, 13, 16, 28] exploit the observation that the order to read chunks is the same as the chunks to be stored for reducing the disk accesses. For example, if container 1 is cached when reading chunk A, then chunk C will hit the cache due to being contained in container 1, avoiding to re-access the disk. Moreover, the information of next required chunks can be obtained via recipes in advance, and thus some schemes adopt a look-ahead window to assemble the chunks belonging to the same container [9, 20], which avoid the frequent accesses to the same container. However, the fragmentation problem is exacerbated over time, since the fragmented chunks are generated during the deduplication process as shown in Figure 2, while the caching schemes don't modify the deduplication process.

A more promising way to improve the restore performance is to enhance the physical locality of the backup streams by rewriting some duplicate chunks. For example, we only need to read 4 containers for rewriting and storing the chunks of the third backup stream together, which however reads 6 containers in Figure 2. Various rewriting approaches propose different measures to determine which chunks to be rewritten, such as Content-Based Rewriting algorithm (CBR) [16], Chunk Fragmentation Level (CFL) [27], Capping [20] and other variants of capping [8, 34]. Moreover, Fu et al. [13] exploit the historic information to rewrite the chunks. Existing rewriting schemes rewrite the chunks meeting some fragmentation quantization standards. However, these rewriting schemes decrease the deduplication ratios due to the existence of duplicate chunks, which consume a large amount of available space.

3 Observations on Fragmented Chunks

The above analysis show that the fragmented chunks are generated in the deduplication phase. In order to enhance the physical locality, our work modifies the



(a) Linux Kernel. **(b)** Gcc. **(c)** Fslhomes. **(d)** Macos.

Figure 3. The chunk number distribution over different backup versions.

deduplication process to store the chunks. To gain more insights about how the fragmented chunks are generated among different backup versions, we count the chunk numbers of different backup versions on various workloads, including Linux kernel, gcc, fslhomes and macos. All these workloads are generally used in the deduplication systems, and more details about the used workloads are shown in Section 5.

The heuristic experiment is conducted on Destor [1] which is a widely used deduplication platform. We assign an infinite buffer to store the information of the chunks without modifying other components of Destor. The assigned buffer is used to store the metadata (including fingerprint, chunk size and version tag) of all chunks, where the version tag indicates the backup version recently containing the chunk. For example, the version tags of all chunks are set to $V1$ when we deduplicate the first backup version. When we next process the second backup version, the duplicated chunks hit the buffer and we modify the version tags of these chunks to $V2$, which indicates that these chunks are recently contained in version 2. Moreover, the unique chunks in the second backup version are stored in the buffer with the version tag $V2$, while the remaining chunks (i.e., not duplicate with the second backup version) in the buffer keep the version tag $V1$. The subsequent backup versions are processed in the same way. As a result, the chunks that are not contained in the new backup versions will always maintain the old version tags.

We count the different chunks according to the version tags after processing each backup version, and the results are shown in Figure 3. As shown in Figure 3a, there are 1,557 $V1$ chunks after deduplicating the first backup version, which decreases to 734 after processing the second backup version and no longer decreases in subsequent backup versions. Such results indicate that these 734 chunks don't appear in the second backup version and the subsequent backup versions. We have the same observation on other chunks and workloads, as shown in Figures 3b and 3c. The observation on macos is a little different, as shown in Figure 3d. For example, the $V1$ chunks not only decrease in the second backup version, but also decrease in the third backup version. After processing the subsequent two backup versions, $V1$ chunks hardly decrease.

From the above results, we observe that the chunks not appearing in the current backup version have a low probability to appear in the subsequent backup versions. The obtained observation offers insights in the real-word applications, since the new backup version is generated via upgrading the old ones, which contains most contents of the old versions and generates some new data. Besides, some existing works [27–29] demonstrate that newer backup versions are more likely to be restored from the system crashes or version rollbacks than the old ones.

Based on these observations, we hence attempt to ensure the restore performance for the new backup versions, e.g., storing the $V8$ chunks closely to improve their physical locality. The proposed scheme can be viewed as a reverse online deduplication process, i.e., the fragmented chunks are generated in the old backup versions rather than the new ones. Through this way, we store the chunks of the new backup version closer without rewriting the duplicate chunks like the traditional schemes. Moreover, the restore performance of the new backup version is improved without decreasing the deduplication ratio.

It is worth noting that all the observations come from backup systems, e.g., backup the different versions of the software (such as gcc, linux kernel) and the snapshots of the file systems. We have the similar observations on other workloads (e.g., gdb, cmake).

4 The Design of HiDeStore

In this section, we demonstrate the design of HiDeStore, a backup system with high deduplication and restore performance. To efficiently store the chunks and improve the restore performance, we modify both the deduplication and restore phases. One of the key insights is to only search the chunks with high probability (i.e., the previous backup versions) to be deduplicated with coming chunks. Another insight is to classify and respectively store the hot and cold chunks, which groups the chunks of new backup version closely to alleviate the chunk fragmentation.

Figure 4 illustrates the system overview of HiDeStore. In the deduplication phase, the backup streams are processed via chunking and hashing like traditional deduplication schemes. One difference is that HiDeStore only searches the fingerprint cache without further searching the full index table on disks. The chunks matching the fingerprint cache are duplicate, otherwise unique. In this way, HiDeStore significantly reduces the memory consumption and avoids the expensive disk access. To achieve a high deduplication ratio, the fingerprint cache mainly contains the chunks with a high duplicate probability according to the observation from Figure 3. More details are elaborated in Section 4.1.

To alleviate the chunk fragmentation for new backup versions, HiDeStore respectively stores the hot and cold chunks, which works like a filter as shown in Figure 4. The contents of identified unique chunks are not directly stored in the containers as traditional deduplication schemes. Instead, the coming unique chunks are temporarily stored in the active containers, which are considered as hot chunks. After processing one backup version, the chunks not appearing in the current version are kicked out from the active containers and stored into the archival containers, considered as cold chunks. In the context of our paper, the archival containers are the same as those in traditional deduplication systems for archival purpose. The active containers change frequently, which insert the hot chunks and remove the cold ones. More details are shown in Section 4.2.

During the process of moving chunks from active containers to archival containers, the recipe is updated to record the locations of chunks for future data recovery. To reduce the overhead for updating recipes, we only update the previous recipe rather than all recipes. We also periodically eliminate the dependency among recipes to reduce the overheads of reading recipes, as shown in Section 4.3. The details of restoring and removing expired backup versions are shown in Sections 4.4 and 4.5.

Our proposed scheme is efficient for archival backup systems, and there exist many redundant data among different versions. For example, the newer version of the software is upgraded from the old one, thus containing many duplicate data with the old version. For the workloads that are not included in this paper, we simply trace the chunk distribution among versions and determine whether to use the proposed scheme, which produces low overhead since we only need to trace the metadata of the chunks.

4.1 Fingerprint Cache with Double Hash

The observation from Figure 3 indicates that the chunks not appearing in current backup version (say cold chunks) have a negligible probability to appear in subsequent backup versions. Therefore, we don't need to search these cold chunks when deduplicating the subsequent backup versions.

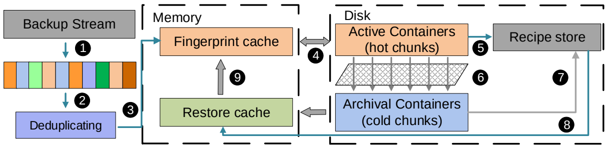


Figure 4. System overview of HiDeStore.

An intuitive design is to only deduplicate the current backup version against the one with high duplicate probability. However, directly using the traditional fingerprint cache in HiDeStore incurs two issues. First, the cache unit of the traditional fingerprint cache is a container, which may contain the cold chunks and consume a large amount of the available space. As a result, the limited cache fails to provide enough space for the hot chunks, and thus the deduplication ratio decreases. Second, the traditional fingerprint cache fails to identify whether the chunks are hot or cold. As a result, we can't separately store the different chunks and enhance the physical locality.

To address these issues, we propose to use two hash tables in the fingerprint cache, respectively represented as $T1$ and $T2$ as shown in Figure 5. In both hash tables, the fingerprints and metadata of the chunks are respectively served as keys and values, where the metadata consist of the chunk size and the ID of active containers that contain the chunks (abbreviated as ***CID***). At the beginning to deduplicate the current backup version (represented as CV), $T1$ contains the metadata of the chunks in the previous backup version (represented as PV) and $T2$ is empty. During the deduplication phase, $T2$ is used to contain the chunks of CV. Both the new unique chunks and the chunks hitting the fingerprint cache are inserted into $T2$. The chunks that hit $T1$ are removed from $T1$ at the same time. After processing CV, the chunks remaining in $T1$ are cold chunks, while the chunks in $T2$ are hot chunks.

Figure 5 illustrates the workflow of the proposed fingerprint cache. There are three cases to deduplicate the coming chunks, which sequentially search $T1$ and $T2$ to identify the duplicate and unique chunks. In Case one, chunk A is considered as a unique chunk, since both $T1$ and $T2$ don't have a duplicate chunk with A. Hence, we insert the metadata of chunk A into $T2$, and store the content of chunk A into an active container, as shown in Section 4.2. In Case two, chunk B is considered as a duplicate chunk due to finding a match in $T1$. In this case, we don't need to store the content of chunk B since it has been stored. However, the metadata of chunk B is removed from $T1$ and inserted to $T2$, indicating that chunk B is a hot chunk and has a high probability

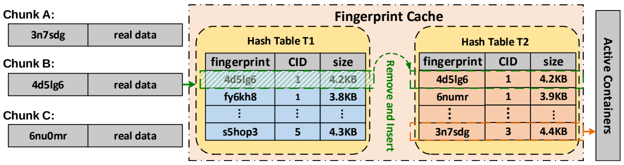


Figure 5. The structure of the fingerprint cache.

to appear in the subsequent backup versions. In Case three, chunk C is also identified as a duplicate chunk due to matching in $T2$. Unlike chunk B, we don't need to execute any operations on chunk C, since its metadata has been stored in $T2$.

After deduplicating CV, the chunks not appearing in CV are left in $T1$, which become cold chunks. The chunks in $T2$ are hot chunks and may appear in the subsequent backup versions. After separating the hot and cold chunks, HiDeStore moves the cold chunks from active containers to archival containers after deduplicating CV and updates the recipes, as shown in Sections 4.2 and 4.3. Finally, we destroy $T1$ and $T2$. When deduplicating the next backup version, the metadata of CV in the recipe is prefetched to $T1$ to serve as PV.

In the hash table, we use fingerprint of each chunk as a key, which is calculated via SHA-1. The probability of a hash collision is much smaller than that of a hardware error [41]. For the case of macos in Figure 3d, we add another hash table to classify the hot and cold chunks, and the deduplication process is similar with Figure 5. The sizes of $T1$ and $T2$ are bounded and hardly full, since the hash tables only contain the metadata and the total size of a hash table is limited to the size of one (or two) backup version(s). Take the data in macos (a very large workload) as an example, one version contains about 5 million chunks, and the total size of $T2$ is more than 100MB (5,000,000*28byte), where 28-byte data consists of 20-byte fingerprint, 4-byte ID, 4-byte size, as shown in Figure 5.

Unlike the traditional deduplication schemes (i.e., locality- and similarity-based schemes), HiDeStore only searches the fingerprint cache, which significantly improves the deduplication throughput. At the same time, HiDeStore achieves almost the same deduplication ratio like the exact deduplication schemes, as shown in Section 5.2. The benefit mainly comes from the observation from Figure 3, i.e., searching the hot chunks is enough since the cold chunks have a negligible probability to appear in subsequent backup versions.

4.2 Chunk Filter to Separate Chunks

In the traditional deduplication systems, the identified unique chunks are directly written to containers according to the order of the arriving chunks, which however incurs the chunk fragmentation, as shown in Figure 2.

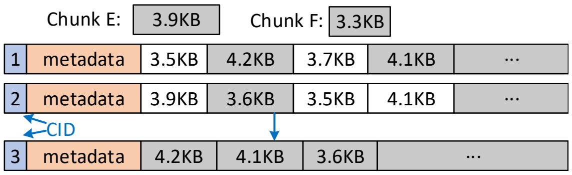


Figure 6. Compact sparse containers.

Unlike them, HiDeStore separately stores the hot and cold chunks by changing the writing paths of the chunks.

Figure 6 shows the common structure of a container, including metadata and the real data of the contained chunks. The metadata consists of container ID, the data size and a hash table for the stored chunks, where the key is a fingerprint and the value is a pointer to the corresponding chunk. We use two kinds of containers to store the hot and cold chunks, including *active containers* and *archival containers*. The active containers are dynamic due to frequently inserting and removing chunks, while the archival containers are static unless removing expired data. Each container is 4MB, i.e., the same size as the traditional containers.

When storing the contents of the chunks, the hot chunks are temporarily stored in active containers during the deduplication phase. After deduplicating one backup version, the chunks remaining in $T1$ are considered to be cold, which are moved to archival containers. The process of moving chunks from active containers to archival containers works like a filter, as shown in Figure 4. After removing the cold chunks, we need to compact the active containers to enhance the physical locality. Otherwise the new chunks are scattered into more containers and the fragmentation is exacerbated. However, the space of the removed chunks can't be directly reused due to the unequal sizes. For example, the deduplication systems generally use content-based chunking algorithms to avoid the boundary-shift problem [26, 40], which generate variable-length chunks. As shown in Figure 6, the chunks with 3.5KB and 3.7KB are removed from an active container, releasing 7.2KB in total. However, the coming chunk E with 3.9KB can't be inserted into this container since the free space is not continuous. Although chunk F with 3.3KB can be inserted into the container, more fragmented spaces are generated and wasted.

To reuse the free space in active containers and enhance the physically locality, HiDeStore merges and compacts the sparse containers. We calculate the container utilization (i.e., the total size divided by the used size) to indicate whether the container is sparse or not. Figure 6 illustrates the compaction operation in HiDeStore. The chunks in two (or more) sparse containers are written into the same container without considering the order, since all these chunks are hot chunks, which will be prefetched together during reading. To improve the space utilization,

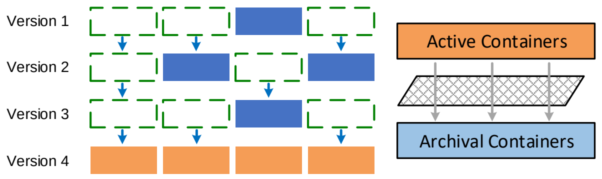


Figure 7. Update recipes. *The blue and orange chunks are respectively stored in archival and active containers.*

the compacted container is stored on disk by overwriting the sparse container whose CID is the smallest.

Although moving the cold chunks from active containers to the archival ones incurs some overheads as shown in Section 5.4, we show that such moving process can be done offline, since the total deduplication process is implemented as a multi-thread pipeline. Once the hot and cold chunks are classified in the hash tables, the next backup version can be processed. Moreover, compared with the traditional deduplication schemes, HiDeStore significantly mitigates the chunk fragmentation for new backup versions, since the chunks of new backup version are stored closely in the active containers, rather than physically scattered to different locations.

4.3 Update Recipes

In the deduplication systems, the recipe is generally used to record the metadata of chunks in a backup stream, which shows how to restore the original data. The metadata consists of the fingerprint, chunk size and ID of the container that contains the chunk (represented as CID). We need to update the recipes when moving the cold chunks, since the locations of the chunks are modified. However, we have no prior knowledge about which recipes contain these cold chunks unless checking all the recipes, which incurs high overheads. Moreover, each recipe needs to be updated multiple times as long as the recipe contains the moved cold chunks.

To reduce the overhead for updating recipes, we only update the previous recipe of the current backup version rather than all the recipes. For the case of macos, we update the last two recipes. Figure 7 illustrates the principle of updating recipes after deduplicating backup version V_4, and the corresponding recipe is represented as R_4. We record the CIDs of all chunks as 0 in R_4 to indicate that all chunks are stored in active containers, and the specific active container is obtained via checking the fingerprint cache. When the cold chunks are moved from active containers to the archival ones, we update R_3 by modifying the CIDs of these cold chunks to the corresponding archival container IDs (i.e., contain the chunks). The CIDs of the remaining chunks in R_3 are modified to the negative ID of V_4. For example, the value 4 indicates that the chunk is stored in the archival container 4, while the value -4 indicates that we need to further check R_4 to find the chunks.

Algorithm 1: Update Recipes

```
Input: Recipe $R[N]$, Hash Table $T$
Output: Updated Recipe $R[N]$
foreach chunk in R[n-1] do
    if chunk.CID>0 then
        insert chunk into T
    end
end
while previous recipe R[n-1] exist do
    foreach chunk in R[n-j] do
        if chunk.CID<0 then
            if chunk have a match p in T then
                chunk.CID=p.CID
            else
                chunk.CID=-n
            end
        end
        if chunk.CID>0 then
            insert chunk into a new Hash Table t
        end
    end
    HashTableDestroy(T) and T=t;
end
```

As a result, all recipes form a chain, as shown in Figure 7. We need to read multiple recipes to find the concrete location for each chunk when restoring the old backup version, which incurs long latency. Instead, we periodically update the recipes before restoring. Algorithm 1 shows an example for updating all the recipes. We read recipe R_{n-1} with the assumption that the newest backup version is n, and insert the chunk whose CID is larger than 0 into hash table T, which is used to update the previous recipe R_{n-2} (Lines 1-5). The negative CID is modified when the chunk has a match in the hash table T (Lines 9-10). The other negative CIDs are modified to $-n$ (Lines 11-12), indicating the chunks are stored in active containers. At the same time, the chunks whose CIDs are larger than 0 are inserted into a new hash table t (Lines 15-17). Finally, the hash table t is used to update the previous recipe, R_{n-3} (Line 20).

The hash table uses the fingerprint as a key and the pointer to the chunk as a value. The recipe is updated from the last modified version. For example, recipe R_1 has been pointed to R_4 in the last updating operation, and we directly read R_4 to update R_1 next time. The recipe only records the metadata of the chunks, which is easy to be updated with low overhead, as shown in Section 5.4. Moreover, we only need to update the recipe as needed offline, and thus the overhead of accessing the recipes is negligible.

4.4 The Restore Phase

The data are restored according to the recipes. In the traditional deduplication systems, all CIDs in recipes are positive, indicating which containers contain the real data of chunks. Unlike them, the recipes in HiDeStore contain 3 types of CID. The positive CID indicates the

Table 1. Characteristics of workloads

Dataset	Kernel	gcc	fslhomes	macos
total size	64GB	105GB	920GB	1.2TB
total versions	158	175	102	25
Deduplication ratio	91.53%	78.75%	92.17%	89.56%

archival container, the negative CID indicates the backup version and 0 indicates the active container.

After reading the recipes, the data can be easily restored via traditional caching schemes, such as chunk-based or container-based caching schemes [9, 13, 16, 28], and forward assemble schemes [9, 20]. By storing the hot chunks together, the fragmentation of new backup versions is alleviated, and the restore performance is improved.

4.5 Removing Expired Versions

In the deduplication systems, the expired versions are removed for space savings [13, 29]. However, we can't directly remove all the chunks of the expired version, since some chunks may also belong to other backup versions. We need to detect the chunks that only belong to the expired version before being removed, which however incurs high overhead due to the need for checking all the backup versions. Moreover, the chunks of different versions are interleaved together, as shown in Figure 2, requiring some garbage collection efforts to reclaim the space for the deleted chunks.

In practice, all efforts for chunk detection and garbage collection are easy in HiDeStore, since HiDeStore has stored the chunks belonging to different backup versions separately. For example, the chunks only belonging to V_1 are stored in archival containers. No subsequent backup versions refer to these chunks according to the observation from Figure 3. We directly remove these chunks when removing expired V_1 without the needs for expensive chunk detection and garbage collection.

5 Performance Evaluation

In this section, we evaluate the performance of HiDeStore on different workloads. Since the traditional deduplication schemes separately achieve high performance in terms of deduplication and restore, we respectively select the state-of-the-art schemes for comparisons.

5.1 Experimental Setup

We implement a prototype of HiDeStore based on Destor [14], which is a widely used deduplication framework and supports the general deduplication pipeline procedure, including chunking, hashing, indexing, rewriting and storing. The chunking phase uses the TTTD chunking algorithm [12] and the hashing phase uses SHA-1 to calculate the fingerprint, due to their simplicity and easy-to-use strength. The hash tables in HiDeStore use the fingerprints as the keys, which rarely have hash collisions [31].

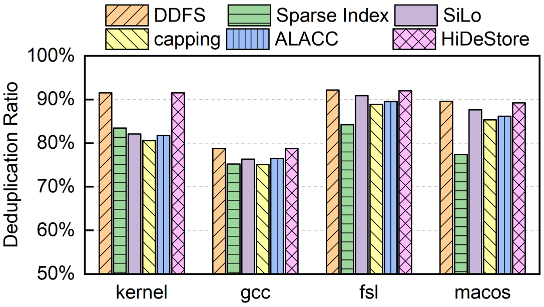


Figure 8. The deduplication ratios.

For the deduplication performance, we compare HiDeStore with DDFS [41], Sparse index [21] and SiLo [35]. DDFS is an exact deduplication system, which achieves the highest deduplication ratio since all the duplicate chunks are removed. Sparse index and SiLo reduce the memory consumption and improve the deduplication throughput by combining locality and similarity. We use the implementations of these schemes from Destor [1]. Moreover, for the restore performance, we compare HiDeStore with caching- and rewriting-base schemes, including capping [20], ALACC [9] and FBW [8]. We use the implementation of ALACC from Ref. [3]. Due to no open source code available of FBW is, we re-implement FBW following the design in the original work [8]. For all schemes, we use the same configurations as the original work to facilitate fair comparisons.

We use four real-world datasets as shown in Table 1, which are widely used in other works [8, 9, 13, 14, 35]. Specifically, kernel [5] and gcc [4] are two generally used public datasets, which consist of the real data of the corresponding software. Fslhomes [2] and macos [2] are two trace datasets, which consist of server snapshots from users. Two consecutive versions are the most similar expect that there are large upgrades, as shown in Figure 3.

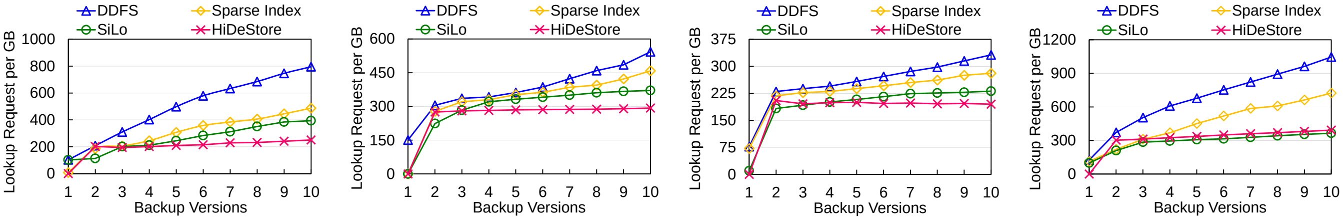
All the experiments run on a Linux server (kernel version v4.4.114), which has two 8-core Intel Xeon E5-2620 v4 @2.10 GHz CPUs (each core with 32KB L1 instruction cache, 32KB L1 data cache, and 256KB L2 cache), 20MB last level cache and 24GB DRAM.

5.2 Deduplication Performance

In general, the deduplication system needs to be examined in three performance metrics, including deduplication ratio, deduplication throughput and memory consumption for index table.

5.2.1 Deduplication Ratio.

To evaluate the deduplication ratio, we divide the



(a) Linux Kernel. **(b)** Gcc. **(c)** Fslhomes. **(d)** Macos.

Figure 9. Lookup overhead among different deduplication schemes.

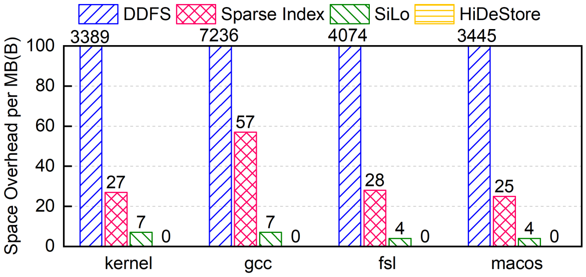


Figure 10. The index table overheads.

eliminated data size by the total size of the dataset, and the results are shown in Figure 8. DDFS achieves the highest deduplication ratio due to using the exact deduplication approach. We observe that HiDeStore achieves almost the same deduplication ratio as DDFS, which is higher than sparse index and SiLo. The main reason is that the chunks with a high probability to be deduplicated are maintained in the fingerprint cache. HiDeStore achieves high deduplication ratio via searching these chunks during the deduplication phase.

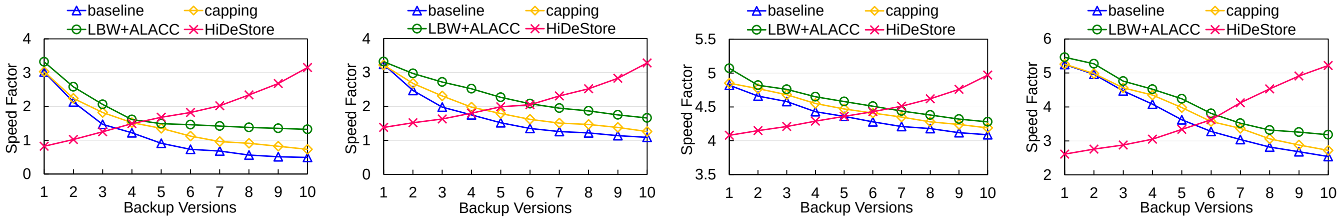
Both sparse index and SiLo decrease some deduplication ratios due to the near-exact deduplication approaches, depending on the sampling methods. The near-exact deduplication schemes group multiple chunks into a segment, and partial chunks are sampled to serve as the feature of each segment. The segment is prefetched for deduplication if its feature is the same, or the most similar, with that of the new segment. However, the sampled features may overlook some chunks which have high duplicate probability, thus decreasing the deduplication ratios. Increasing the sampling ratio can improve the deduplication ratio, which however requires more space to store the features and more time to be spent on searching the features. As a result, we have to make a trade-off between deduplication ratio and deduplication throughput. Based on Figure 3, it is worth noting that if the near-exact deduplication schemes choose the segments of last backup version as the candidates to deduplicate the new segments, the high deduplication ratio is achieved, like HiDeStore.

As shown in Figure 8, we evaluate the deduplication ratios of rewriting schemes, including capping [20] and ALACC [9]. The rewriting schemes are evaluated based on SiLo [35], which achieves good deduplication performance by making a trade-off between deduplication ratio and deduplication throughput. We observe that the rewriting schemes decrease more deduplication ratios than the exact deduplication scheme, due to the existence of duplicate chunks. Moreover, when processing more data, the rewriting schemes rewrite more duplicate chunks to alleviate the chunk fragmentation, which further decreases the deduplication ratios. Even when the deduplication ratio decreases 1%, extra 10GB are required for storing 1TB data.

5.2.2 Deduplication Throughput.

The used experimental platform (i.e., Destor [1]) stores the indexes in a fingerprint cache and a full index table. To simulate the requests to disks, Destor evaluates the number of lookup requests for the full index table, while the lookup requests for unique chunks are eliminated since most are answered by the in-memory Bloom filter [14]. Thus, the index throughput evaluated by Destor is not absolute throughput. We use the same metric as Destor [14] to evaluate the lookup overhead, i.e., *lookup requests per GB*, which is defined as the number of the required lookup requests to the full index table to deduplicate 1GB data. The high lookup requests per GB to the full index table represent the low deduplication throughput, since we need to spend more time on randomly accessing the disks. However, HiDeStore doesn't need to access the full index table during deduplicating, since the chunks that have the high probability to be deduplicated have been stored in the fingerprint cache before deduplication. To facilitate fair comparisons, we use the same unit size of a lookup request as the traditional schemes, and evaluate the number of the lookup requests. The results among different schemes are shown in Figure 9.

As shown in Figure 9a, we observe that HiDeStore achieves the best results among all schemes. Specifically, HiDeStore reduces up to 71% of the lookup requests than DDFS. HiDeStore obtains more benefits when deduplicating more backup versions, since the lookup overhead of HiDeStore is bounded to the size of one



(a) Linux Kernel.

(b) Gcc.

(c) Fslhomes.

(d) Macos.

Figure 11. The restore performance among different schemes.

backup version. Compared with near-exact deduplication schemes, HiDeStore respectively reduces the lookup overhead by up to 48% and 38% than sparse index and SiLo. The main reason is that HiDeStore only needs to search the fingerprint cache without the needs to frequently access the full index table on disks. However, the traditional deduplication schemes need to access the full index table when the coming chunks miss in the fingerprint cache.

Moreover, we observe that the lookup overheads of sparse index and SiLo increase slowly with the backup versions. Since these schemes exploit the logical locality, i.e., the segments of the last backup version are most likely to be accessed during deduplication, as shown in Figure 3. Hence, the fingerprint cache has a high hit ratio when most segments of the last backup version are maintained. However, the segments of other backup versions may also be prefetched, since the sampled chunks possibly exist in multiple backup versions. Unlike them, HiDeStore fully exploits the observation from Figure 3 without the needs of sampling the chunks. Only the chunks with high the duplicate probability are accessed, and HiDeStore achieves the best results. We have the same observations on other datasets, as shown in Figures 9b and 9c.

For macos, the lookup overhead of HiDeStore is a little higher than SiLo, as shown in Figure 9d. Since the chunks with the high duplicate probability occur in the last two backup versions, which incurs more lookup requests. However, it is worth noting that HiDeStore prefetches the fingerprint cache before deduplicating each backup version, which doesn't block the deduplicating process like traditional deduplication schemes. All the lookup requests in HiDeStore are sequential, since the metadata of chunks are stored together in the recipe. Hence, HiDeStore delivers high lookup performance in the real-world backup systems.

5.2.3 Space Consumption for Index Table.

The space consumption consists of the index table, the stored data and recipes. The stored data are evaluated by the deduplication ratio, and the higher deduplication ratio indicates the less data are stored, as shown in Figure 8. The recipes record the positions of all the chunks and indicate how to recover the data, and thus the sizes of recipes are constant. Except these two metrics, we use *space overhead per MB(B)* [35] to evaluate the index table overhead, which is defined as the required space for the indexes to deduplicate 1MB data, like existing schemes [14, 35].

The index table overheads among different deduplication schemes are shown in Figure 10. DDFS stores the indexes of all unique chunks, and has the highest memory consumption. Moreover, the index table overhead is high when the datasets contain a large number of small files. Sparse index and SiLo incur lower index table overheads than DDFS, depending on the sampling approaches and ratios. For example, sparse index reduces the memory consumption by nearly 128× if we set the sample ratio to 128 : 1. SiLo samples the minimal fingerprint of a segment, which further reduces the index table overhead. Moreover, the index table overheads increase when processing a large amount of data.

Unlike them, HiDeStore doesn't need extra space to store the indexes, since HiDeStore deduplicates one backup version against its previous one, and the fingerprint indexes of all chunks in previous backup version have been stored in the recipe. We check the recipe of the previous backup version during deduplication, rather than maintaining an index table. Hence, HiDeStore significantly reduces the index table overhead. Moreover, HiDeStore obtains more benefits when deduplicating more backup versions, since no memory space for the index table is needed, i.e., the recipe of last backup version has maintained the indexes for deduplication.

5.3 Restore Performance

Restoring the data needs to read the data from different containers on disks, and the restore performance is significantly influenced if the chunks are scattered into multiple containers. In our evaluation, the sizes of all containers are set to 4MB to facilitate fair comparisons. We use *speed factor*(MB/container-read) as the metric to measure the restore performance, which is defined as the mean data size that is restored per container [8, 20]. Speed factor is widely used in many schemes [8, 9, 13, 16, 20], since this metric avoids the speed variance due to the physical configurations in file systems. Higher speed factor indicates higher restore performance, since the data are physically gathered. The scheme that doesn't rewrite chunks is served as the baseline. The state-of-the-art rewriting scheme is ALACC [9], which achieves the best results among existing schemes when using FBW as the restore caching scheme [8]. Other schemes are implemented based on Destor [1], which uses FAA [20] as the restore caching scheme.

The results of the restore performance among different schemes are shown in Figure 11. We observe that HiDeStore achieves higher restore performance on new backup versions than other schemes, which however sacrifices the restore performance of the old backup versions. Specifically, HiDeStore improves the restore performance by up to 1.6× than ALACC on the new backup versions. The main reason is that HiDeStore groups the chunks belonging to new backup versions closely in the active containers, and stores the identified cold chunks in the archival containers. These cold chunks may belong to multiple old backup versions, and the fragmentation of old backup version is exacerbated over time. However, the fast restore performance for new backup version is more important than the old ones, since a number of existing works [27–29] demonstrate that newer backup versions are more likely to be restored.

From Figures 8 and 11, we observe that HiDeStore keeps high restore performance for the new backup versions without decreasing the deduplication ratio. However, the rewriting schemes have to rewrite more and more chunks to alleviate the fragmentation over backup versions, which consume more space to store the duplicate chunks. In order to achieve high restore performance, existing rewriting schemes have to set a low rewriting-threshold to rewrite more chunks, which further consumes more space.

5.4 HiDeStore Overheads

The overheads of HiDeStore come from two aspects, including updating recipes and moving the chunks from active containers to archival containers.

We update the recipes in two cases. First, after processing one backup version, we only update the previous recipe during the deduplication phase. Second, when restoring one backup version, we update the recipe according to Algorithm 1. The mean latency for updating one recipe depends on the sizes of the workloads, as shown in Figure 12. For example, 21ms is required to update one recipe of Linux Kernel. We observe that the overhead of updating one recipe is low due to the small size of the recipe. Moreover, Algorithm 1 is carried out offline before restoring the data, and the overhead is negligible.

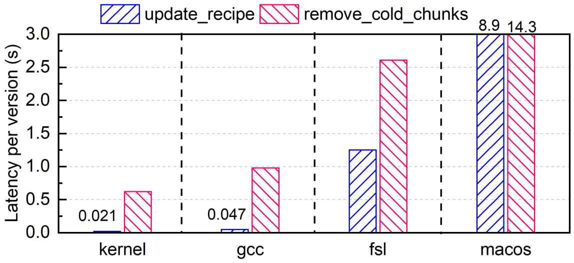


Figure 12. HiDeStore running overheads, *including the processes to move cold chunks and update recipes.*

The process of moving chunks is carried out after deduplicating one backup version, which can be processed offline due to the pipeline implementation. We evaluate the latency for moving chunks and merging sparse containers, and the results are shown in Figure 12. The chunks in the active containers are sequentially written to archival containers. Hence, the process of moving chunks is not influenced by the chunk fragmentation.

5.5 The Deletion

The expired backup versions need to be removed to save space [13, 29]. Unlike the traditional schemes, HiDeStore separately stores the cold and hot chunks during the backup phase, as demonstrated in Sections 4.2 and 4.5. HiDeStore directly removes the cold chunks of the expired backup versions without the needs to detect which chunks only belong to the expired backup version, since the subsequent backup versions don't refer to these cold chunks. Moreover, HiDeStore doesn't need any efforts for garbage collection, since the removed chunks are stored together in the archival containers, which doesn't cause the fragmentation like traditional schemes. Therefore, the overhead of removing expired backup versions is almost zero, which is much lower than those of the traditional schemes.

6 Related Work

Deduplication Schemes for Index-access Bottleneck. Various approaches are proposed to address the index-access bottleneck by exploiting the logical locality, i.e., sequential chunks in a backup stream have a high probability appearing in another backup stream with the same order. Zhu et al. [41] use an in-memory Bloom Filter and cache a sequence of chunks to speed up the deduplication process. Sparse Indexing [21] uses a sampling approach to avoid the needs for full chunk indexing, which significantly reduces the RAM consumption. ChunkStash [10] uses SSD to deliver high access performance. Since the locality information may become outdated over time, Block Locality Cache (BLC) [23] is proposed to always use up-to-date locality information. Extreme Binning [6] exploits file similarity instead of locality to deliver reasonable throughput for non-traditional backup workloads that have poor locality. To achieve high deduplication throughput at low RAM overhead, SiLo [35, 36] efficiently exploits both locality and similarity.

However, these approaches mainly focus on the deduplication phase, while overlooking the chunk fragmentation problem. Different containers have to be accessed to read the scattered chunks, resulting in poor restore performance.

Restore Schemes for Chunk Fragmentation. Existing approaches that address the chunk fragmentation problem are coarsely classified into two categories, including optimizing the restore cache and rewriting duplicate chunks. Specifically, the caching-based schemes exploit the locality, i.e., the order to read chunks is the same as the chunks are stored. Hence, the container-based caching [13, 16, 28] and chunk-based caching schemes [9, 20] are proposed. Moreover, the data streams are restored in a Forward Assembly Area (FAA) by leveraging the prior knowledge of the recipes [20]. Cao et al. [9] combine FAA and chunk-based caching to improve the restore performance.

The schemes that rewrite some chunks to enhance the physical locality become more promising, since the chunk fragmentation is alleviated. The Content-Based Rewriting algorithm (CBR) [16] determines whether to rewrite the chunks based on the contents of the data stream. Chunk Fragmentation Level (CFL) [27] is an efficient quantitative metric to measure the fragmentation, defined as the optimal chunk fragmentation (i.e., the optimal number of containers to hold the data stream) divided by the current chunk fragmentation (i.e., the actual used containers to hold the data stream). Capping [20] limits the maximum number of containers that is referred by a segment, which ensures that the number of containers referenced by each stream is small. Moreover, a variant capping method based on the minimum submodule method is proposed [34]. Cao et al. [8] optimize the capping algorithm by dynamically setting the maximum number of the referred containers for various workloads. However, these rewriting schemes reduce some deduplication ratios due to the existence of duplicate chunks.

Other Schemes for Deduplication. To avoid the boundary shift problem, many content-based chunking schemes are proposed, including TTTD chunking [12], Rabin-based CDC [26], FastCDC [37] and AE [40]. To remove the expired backup versions, some chunk detection schemes [33] and garbage collection schemes [15] are proposed.

7 Conclusion

Data deduplication as an efficient middleware becomes important to obtain space savings, which however suffers from the chunk fragmentation over time. Existing caching- and rewriting-based schemes become inefficient when processing a large number of versions due to the exacerbated fragmentation problem. As a result, it is necessary to obtain a trade-off among deduplication ratio, deduplication throughput and restore throughput. Our proposed HiDeStore explores and exploits the access pattern during deduplication process. We identify the hot and cold chunks, and store the hot chunks closely to enhance the physical locality for the new backup versions. The experimental evaluations show that our proposed HiDeStore improves the deduplication throughput and restore throughput than state-of-the-art schemes, while maintaining a high deduplication ratio with acceptable overheads. We have released the open source code of HiDeStore for public use in GitHub.

Acknowledgments

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 61772212. We are grateful to our shepherd, Pedro Fonseca, and the anonymous reviewers for their constructive comments and suggestions.

References

[1] 2015. Destor. https://github.com/fomy/destor.

[2] 2016. Traces and Snapshots Public Archive. http://tracer.filesystems.org/.

[3] 2018. ALACC. https://github.com/zhichao-cao/dedup_restore.

[4] 2019. GCC. https://ftp.gnu.org/gnu/gcc/.

[5] 2019. Linux Kernel. https://www.kernel.org/.

[6] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. 2009. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, 1–9.

[7] Robert Birke, Isabelly Rocha, Juan Perez, Valerio Schiavoni, Pascal Felber, and Lydia Y Chen. 2019. Differential Approximation and Sprinting for Multi-Priority Big Data Engines. In *Proceedings of the 20th International Middleware Conference (Middleware)*. 202–214.

[8] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David HC Du. 2019. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *17th USENIX Conference on File and Storage Technologies (FAST)*. 129–142.

[9] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. 2018. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST)*. 309–324.

[10] Biplob K Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory.. In *USENIX annual technical conference (ATC)*. 1–16.

[11] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. 2019. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!. In *2019 USENIX Annual Technical Conference (ATC)*. 647–660.

[12] Kave Eshghi and Hsiu Khuern Tang. 2005. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR* 30, 2005 (2005).

[13] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. 2014. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (ATC)*. 181–192.

[14] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. 2015. Design tradeoffs for data deduplication performance in backup workloads. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*. 331–344.

[15] Fanglu Guo and Petros Efstathopoulos. 2011. Building a High-performance Deduplication System.. In *USENIX annual technical conference (ATC)*.

[16] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. 2012. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR)*. 1–12.

[17] Sobhan Omranian Khorasani. 2018. Removing Bottlenecks in Big Data Processing Platforms. (2018).

[18] Sobhan Omranian Khorasani, Jan S Rellermeyer, and Dick Epema. 2019. Self-adaptive Executors for Big Data Processing. In *Proceedings of the 20th International Middleware Conference (Middleware)*. 176–188.

[19] Yan-Kit Li, Min Xu, Chun-Ho Ng, and Patrick PC Lee. 2014. Efficient hybrid inline and out-of-line deduplication for backup storage. *ACM Transactions on Storage (TOS)* 11, 1 (2014), 1–21.

[20] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST)*. 183–197.

[21] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, and Peter Camble. 2009. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality.. In *Fast*, Vol. 9. 111–123.

[22] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. 2012. SAR: SSD assisted restore optimization for deduplication-based storage systems in the cloud. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*. IEEE, 328–337.

[23] Dirk Meister, Jürgen Kaiser, and André Brinkmann. 2013. Block locality caching for data deduplication. In *Proceedings of the 6th International Systems and Storage Conference*. 1–12.

[24] Dirk Meister, Jurgen Kaiser, Andre Brinkmann, Toni Cortes, Michael Kuhn, and Julian Kunkel. 2012. A study on data deduplication in HPC storage systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 1–11.

[25] Dutch T Meyer and William J Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage (ToS)* 7, 4 (2012), 1–20.

[26] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. 2001. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP)*. 174–187.

[27] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. 2011. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications*. IEEE, 581–586.

[28] Young Jin Nam, Dongchul Park, and David HC Du. 2012. Assuring demanded read performance of data deduplication storage with backup datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 201–208.

[29] Chun-Ho Ng and Patrick PC Lee. 2013. Revdedup: A reverse deduplication storage system optimized for reads to latest backups. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*. 1–7.

[30] Dongchul Park, Ziqi Fan, Young Jin Nam, and David HC Du. 2017. A lookahead read cache: improving read performance for deduplication backup storage. *Journal of Computer Science and Technology* 32, 1 (2017), 26–40.

[31] Sean Quinlan and Sean Dorward. 2002. Venti: A New Approach to Archival Storage.. In *FAST*, Vol. 2. 89–101.

[32] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of backup workloads in production systems.. In *FAST*, Vol. 12. 4–4.

[33] Jiansheng Wei, Hong Jiang, Ke Zhou, and Dan Feng. 2010. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–14.

[34] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. 2017. A cost-efficient rewriting scheme to improve restore performance in deduplication systems. In *Proc. MSST*.

[35] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput.. In *USENIX annual technical conference (ATC)*. 26–30.

[36] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2014. Similarity and locality based indexing for high performance data deduplication. *IEEE transactions on computers* 64, 4 (2014), 1162–1176.

[37] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. Fastcdc: a fast and efficient content-defined chunking approach for data deduplication. In *2016 USENIX Annual Technical Conference (ATC)*. 101–114.

[38] Guangping Xu, Bo Tang, Hongli Lu, Quan Yu, and Chi Wan Sung. 2019. LIPA: A Learning-based Indexing and Prefetching Approach for Data Deduplication. In *2019 35th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 299–310.

[39] Qirui Yang, Runyu Jin, and Ming Zhao. 2019. Smartdedup: optimizing deduplication for resource-constrained devices. In *2019 USENIX Annual Technical Conference (ATC)*. 633–646.

[40] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. 2015. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*. IEEE, 1337–1345.

[41] Benjamin Zhu, Kai Li, and R Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System.. In *FAST*, Vol. 8. 1–14.