Li PF, Hua Y, Cao Q. An enhanced physical-locality deduplication system for space efficiency. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(6): 1361-1379 Nov. 2024. DOI: 10.1007/s11390-023-2646-7

An Enhanced Physical-Locality Deduplication System for Space Efficiency

Peng-Fei Li (李鹏飞), Yu Hua* (华 宇), *Distinguished Member, CCF, Senior Member, ACM, IEEE* and Qin Cao (曹 钦)

Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

E-mail: cspfli@hust.edu.cn; csyhua@hust.edu.cn; qincao@hust.edu.cn

Received July 27, 2022; accepted January 29, 2023.

Abstract An abundance of data have been generated from various embedded devices, applications, and systems, and require cost-efficient storage services. Data deduplication removes duplicate chunks and becomes an important technique for storage systems to improve space efficiency. However, stored unique chunks are heavily fragmented, decreasing restore performance and incurs high overheads for garbage collection. Existing schemes fail to achieve an efficient trade-off among deduplication, restore and garbage collection performance, due to failing to explore and exploit the physical locality of different chunks. In this paper, we trace the storage patterns of the fragmented chunks in backup systems, and propose a high-performance deduplication system, called HiDeStore. The main insight is to enhance the physical-locality for the new backup versions during the deduplication phase, which identifies and stores hot chunks in the active containers. The chunks not appearing in new backups become cold and are gathered together in the archival containers. Moreover, we remove the expired data with an isolated container deletion scheme, avoiding the high overheads for expired data detection. Compared with state-of-the-art schemes, HiDeStore improves the deduplication and restore performance by up to 1.4x and 1.6x, respectively, without decreasing the deduplication ratios and incurring high garbage collection overheads.

Keywords deduplication system, data reduction, space efficiency, physical-locality

1 Introduction

Widely used applications, such as IoT (Internet of Things) embeddings, AI, and cloud computing[1–3], generate a large amount of data and require large-scale storage systems. Backup systems[4–6] store various versions of data for software compatibility and rollback, e.g., different versions of Linux kernels and system snapshots. However, the data contain much redundancy due to the similarity among different backup versions. Data deduplication becomes an efficient technique for different storage systems[7–11] to eliminate duplicate data and save space[12, 13].

Deduplication systems improve storage efficiency via eliminating duplicate data, following the workflow of chunking, fingerprinting, indexing, and further storage managements[12–14]. To detect duplicate data, we divide data streams into 4 KB–8 KB chunks and leverage a cryptographic hash function to calculate fingerprints for the chunks, e.g., SHA-1 (Secure Hash Algorithm)[12] and MD5 (Message Digest Algorithm 5). It has been proved that a hash collision of the used cryptographic hash function is much smaller than that of a hardware error[12]; hence unique chunks have different fingerprints, and are stored in typical 4 MB containers on the persistent storage mediums, such as HDD (hard disk drive) or SSD (solid state drive). The chunk references of the original data streams are stored in the recipes for data restoring.

However, the deduplication systems deliver low restore performance after multiple data versions are

Regular Paper

A preliminary version of the paper was published in the Proceedings of ACM/IFIP Middleware 2020.

This work was supported in part by the National Natural Science Foundation of China under Grant Nos. 62125202 and U22B2022.

*Corresponding Author

stored, due to the severe chunk fragmentation problem[15–17]. Specifically, the identified duplicate chunks are pointed to existing containers, while unique chunks are stored in new containers. As a result, the chunks of a data stream are stored in different containers, incurring lots of expensive I/Os to read data in the persistent storage to restore the original data. The data chunks are severely scattered when more versions are stored. Moreover, it becomes hard to remove the expired versions, since the chunks of different versions are physically scattered and interleaved together, which results in expensive efforts to detect the expired chunks and conduct garbage collection.

To improve the restore performance, some designs leverage caching-based schemes to reduce the amount of container reading, e.g., some chunks[17–19] and containers[15, 16, 20] are cached in the memory for future reading. The main insight is to exploit the cache-friendly locality of the backup stream, i.e., the chunks are stored in the same order as they first appear in the stream. Therefore, the obtained containers have a high probability to contain the subsequent chunks of the same data stream. However, caching-based schemes become inefficient when a large number of backup versions are stored, since the chunks are scattered into more different containers and show poor locality for caching. Unlike the caching-based schemes, some schemes rewrite the duplicate chunks to enhance the physical locality of the data stre am[15, 16, 21, 22], i.e., these schemes rewrite some chunks into the same containers. In this way, fewer containers are read to restore the original data. Although the chunk fragmentation problem is alleviated, the deduplication ratio decreases due to the existence of duplicate chunks. Even if the deduplication ratio decreases by 1%, 40 GB extra space is consumed for 4 TB unique data to store the rewritten data, which significantly decreases the storage efficiency.

To remove expired data, existing schemes leverage the reference management approaches to detect the expired chunks, and such schemes need to carefully maintain the reference counters to prevent errors, e.g., the removed chunks are referred by the non-expired backup versions. Moreover, the sparse containers occur after the expired backups are removed, incurring expensive overheads for garbage collection.

Unlike existing schemes, we propose to enhance the physical locality of the backups for better deduplication, restore, and expired data deletion performance. We explore and exploit the behaviors of the fragmented chunks via a heuristic experiment, which traces the storage path and reference patterns of different chunks among various backup versions. We observe that high redundancy arises between adjacent backup versions, and the chunks not appearing in current backup version have a low probability to appear in the subsequent backup versions. Moreover, in backup systems, newer backup versions are more likely to be restored than older versions[20, 22, 23], which implies that the high restore performance of newer backup versions is more important than that of older ones.

Based on the observations, we propose an efficient deduplication scheme with high restore performance and deduplication ratios, called HiDeStore①. The main insight is to classify the hot and cold chunks during the deduplication phase, and store hot and cold chunks in active and archival containers respectively to enhance the physical locality. The hot chunks are referred by subsequent backup versions, while the cold chunks have a low probability to appear in the new backup versions. Based on the high physical locality of different chunks, HiDeStore reads hot chunks to restore the new backups, while directly removing cold chunks for expired version deletions.

Specifically, the workflow of HiDeStore consists of three steps. 1) Hot and cold chunks are classified via the double-hash based fingerprint cache. 2) The contents of different chunks are filtered and stored in active and archival containers, respectively. 3) The recipes are updated for restoring the original data. We construct a recipe chain to reduce updating overheads, and further optimize the process of recipe searching by periodically eliminating the dependency among recipes. Compared with state-of-the-art deduplication schemes, HiDeStore reduces the index lookup overheads by 38% and improves the restore performance by up to 1.6x. By leveraging the Isolated Container Deletion Algorithm (ICDA), HiDeStore becomes efficient to remove expired versions without expensive garbage collection efforts, since the expired chunks are gathered together in archival containers.

This paper has made significant improvements over the preliminary version[24] as the follows.

• *Tracing Storage Patterns of Different Chunks.* We conduct heuristic experiments on multiple workloads to analyze the storage patterns of chunks in

①The source code of HiDeStore is available at https://github.com/iotlpf/HiDeStore, Nov. 2024.

backup systems. The obtained observations motivate us to construct the efficient deduplication system via enhancing the physical locality for different chunks.

• *High Deduplication Performance with High Deduplication Ratios*. We explore the workload characteristics in backup systems and only cache fingerprints of hot chunks for index searching, which avoids frequently accessing disks and achieves high deduplication performance.

• *High Restore Performance for New Backup Versions*. Our proposed HiDeStore filters and stores cold and hot chunks in different containers to enhance the physical locality, which achieves high restore performance for new backup versions, since HiDeStore reads fewer containers than existing schemes.

• *Low Overheads to Remove Expired Backups*. We analyze the processes of removing expired data in existing schemes and observe that the schemes incur high overheads in expired data detection and garbage collection. Therefore, we present an isolated container deletion algorithm to enable HiDeStore to detect and remove expired containers with low overheads.

• *Confirm Observations with Widely Used Datasets*. We add a widely used dataset, Boost[4, 15, 18], to confirm our observations in the backup systems, and obtain the same observation with other datasets, i.e., the adjacent versions are the most similar. Based on the observations, HiDeStore efficiently identifies and stores different chunks for high physical locality.

• *Comprehensive Evaluations*. We conduct evaluations on five widely used datasets to show the strengths of HiDeStore over existing schemes in terms of redundant data deduplication, original data restoring, and expired data deletion.

2 Background

2.1 Workflow of a Deduplication System

Chunk-based deduplication becomes an efficient technique for backup systems to improve the space utilization efficiency[7–11]. In this paper, we focus on the in-line deduplication[13–16, 18, 20, 25–26], i.e., the data is deduplicated once it is stored.

The workflow of a deduplication system is shown in Fig.1. 1) The coming data stream is divided into multiple chunks (e.g., on average 4 KB–8 KB[13]) via various chunking algorithms, such as TTTD (the Two Thresholds, Two Divisors Algorithm) chunking[27], Rabin-based CDC (Content-Defined Chunking)[9], and FastCDC[28]. 2) 20-byte fingerprints are calculated for the obtained chunks via a secure hash function, e.g., SHA-1[12]. It is worth noting that the probability of a hash collision is much smaller than that of a hardware error[12]. 3) The chunks with identical fingerprints are duplicate. Some fingerprints are maintained in the fingerprint cache to accelerate the index searching[13, 14, 29, 30]. 4) When the coming fingerprints miss in the cache, the fingerprints are further searched in the whole fingerprint table on disks to achieve high deduplication ratios. 5) The unique chunks are stored into typical 4 MB containers. The references (i.e., the fingerprints, chunk sizes, and container IDs) of all chunks are recorded in a recipe[13] for the data recovery. The data are restored from system crashes or version rollbacks[16, 17]. 6) To restore the original data, we read the recipe and obtain the recorded chunk references. 7) Chunks are read according to the recipe and the original data are assembled in a chunk-by-chunk manner.

2.2 Fingerprint Access Bottleneck

In the deduplication phase, we search existing fingerprints to identify whether the coming chunks are duplicate. However, the number of fingerprints proportionally increases with the stored data and the fingerprint table possibly overflows the limited memory, e.g., indexing 4 TB unique chunks requires at least 20 GB to store the fingerprints. Therefore, the finger-

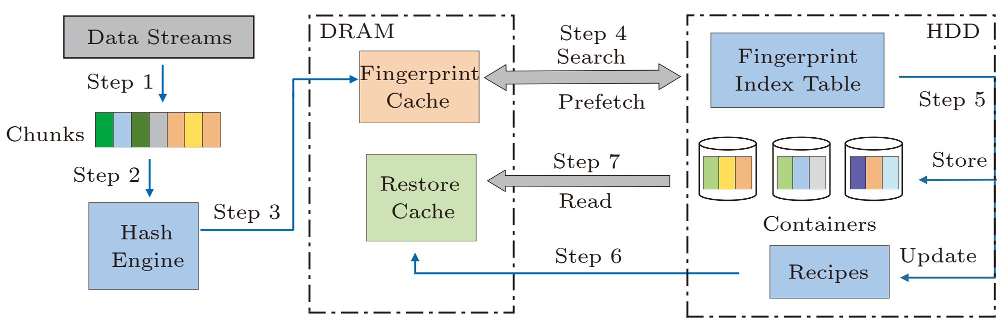

Fig.1. Workflow of a deduplication system.

print access bottleneck occurs when the fingerprint table on disks is frequently accessed, which significantly decreases the deduplication performance[13, 14].

Existing deduplication systems leverage various approaches to improve the hit ratio of the fingerprint cache and avoid expensive I/Os on the disk. Some schemes[13, 14, 25, 26] 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. Thus, 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[14, 31]. For the workloads that have little or no locality, similarity-based approaches were proposed[29, 30] for better prefetching. However, we have to make a trade-off between deduplication ratios and throughput, since the efficiency of existing schemes depends on the locality and similarity of the workloads. Moreover, these schemes overlook the chunk storage management during the deduplication phase, and incur the severe chunk fragmentation problem over time, as shown in Subsection 2.3.

2.3 Chunk Fragmentation Problem

The restore phase reads chunks from different containers according to the recipe, and assembles the original data chunk by chunk. However, the restore performance suffers from the chunk fragmentation problem[6, 15–18, 21], i.e., the chunks of the same data stream are scattered into various containers, incurring frequent disk accesses during the recovery phase. The main reason is that the identified duplicate chunks are not stored together with unique chunks when a data stream is processed.

Fig.2 illustrates how the chunk fragmentation problem arises with the assumption that each container contains at most three chunks. During the deduplication phase, the unique chunks are stored in containers when the chunks 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, and H) are not stored, while the unique chunks (e.g., chunks I, J, K, and L) are stored in containers 4 and 5. As a result, we need to access five containers to restore the second backup stream. The same deduplication mechanism is applied to the third data stream, and we need to access six different containers to restore the third data stream. Such chunk fragmentation problem is exacerbated over time when more backup versions are stored.

Some schemes are motivated from the observation that the order to read chunks is the same as that to store the chunks, and they propose caching-based schemes to improve the restore performance. Hence, we cache a sequence of chunks in one disk access to speed up the chunk reading. For example, if container 1 is cached when chunk A is read, chunk C will hit the cache since chunk C has already been contained in container 1, avoiding re-accessing the disk. Moreover, some schemes propose a look-ahead window to assemble the chunks belonging to the same container[17, 18], which avoids the frequent accesses to the same container. However, the chunk fragmentation problem is exacerbated when more backup versions are stored, since the chunks are scattered into a large number of containers and exhibit poor physical locality.

A more promising way to improverestore performance is to enhance the physical locality of the backup streams by rewriting some duplicate chunks. For example, we only need to read four containers when the chunks of the third backup stream are stored together, rather than reading six containers in Fig.2.

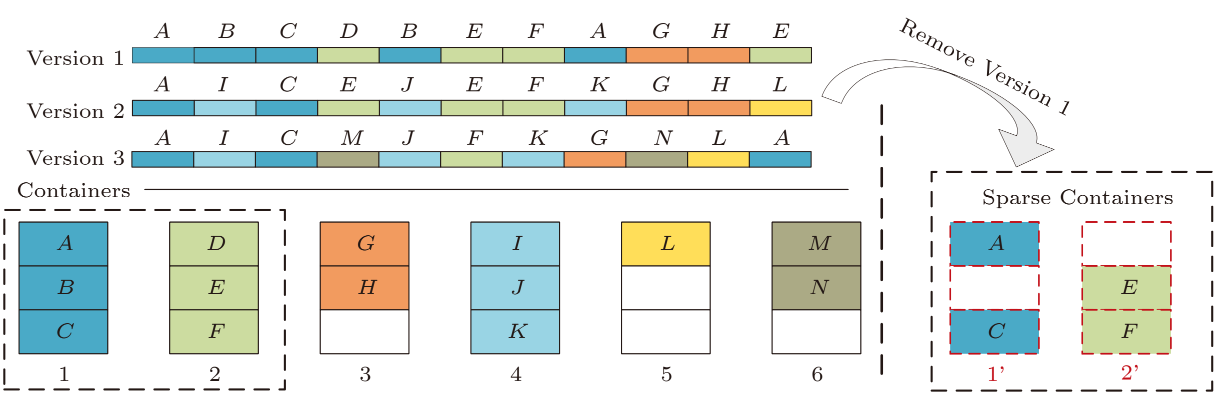

Fig.2. Chunk fragmentation problem[24]. The order of versions is determined by the generation time.

Various rewriting schemes leverage different approaches to determine which chunks to rewrite, such as the Content-Based Rewriting algorithm (CBR)[16], Chunk Fragmentation Level (CFL)[22], and capping-based schemes[17, 21]. Moreover, Fu *et al.*[15] exploited the historic information to rewrite the chunks. However, these rewriting schemes decrease the deduplication ratios due to the existence of duplicate chunks, and the duplicate chunks consume much available space. For example, 40 GB of extra space is consumed for 4 TB of unique data to store the rewritten data even if the deduplication ratio decreases by 1%.

2.4 Garbage Collection

Physical fragmented chunks often result in high overheads for garbage collection when expired versions are removed, due to the time-consuming phase of identifying the chunks that are only referred by the expired backups. Moreover, the chunks of different versions are interleaved together, requiring many garbage collection efforts to reclaim the space for the deleted chunks. As shown in Fig.2, only chunks B and D are removed when backup version 1 is removed, since only chunks B and D are not referred by other versions. However, identifying chunks B and D becomes a bottleneck due to the complicated reference management for chunks. Moreover, removing fragmented chunks results in sparse containers, such as containers 1' and 2' in Fig.2, and these sparse containers waste much storage space.

In order to remove expired backups, existing approaches leverage offline and inline algorithms for backup deletions[15, 32, 33]. For example, all fingerprints of chunks are traversed when the system is idle, and additional metadata for the chunk references is maintained during the deduplication phase. However, these approaches incur high time and space overheads, due to the needs of managing the metadata of chunk references. Furthermore, extra efforts are consumed on merging the sparse containers after the expired backups are removed.

3 Observations on Fragmented Chunks

To gain more insights about the fragmented chunks, we conduct a heuristic experiment on five widely used datasets, including Linux Kernel[4], GCC[15], Fslhomes[18], MacOS[21], and Boost. More details about the used workloads are shown in Section 5. The heuristic experiments aim to obtain the patterns of chunk references among different backup versions, where the chunk reference points to the container that contains the corresponding chunk.

We conduct a heuristic experiment based on a widely used deduplication platform, called Destor[4]. Specifically, we assign an infinite buffer to store the metadata of chunks, including fingerprints, chunk size, and a version tag, where the version tag indicates the most recent backup version containing the chunk. For example, the version tags of all chunks are set to V1 when the first backup version is deduplicated. When the chunks of the second backup version have matches within the buffer, we modify the version tags of these chunks to V2 to indicate that these chunks are contained in the backup version 2. At the same time, the unique chunks in the second backup version are stored in the buffer with the version tag V2. The remaining chunks (i.e., not appearing in the second backup version) in the buffer keep the version tag V1, indicating that these chunks are contained in the backup version 1. The heuristic experiment processes all data in the same way. After all backup versions are processed, the version tags indicate the newest backup versions containing the chunks.

To figure out the reference patterns of different backup versions, we count the numbers of various version tags after each backup version is processed, and the results are shown in Fig.3. As shown in Fig.3(a), there are 1 557 V1 chunks after the first backup version is deduplicated. The number of V1 chunks decreases to 734 after the second backup version is processed and almost no longer decreases in subsequent backup versions. Such results indicate that these 734 chunks are not contained in the subsequent backup versions, which incurs chunk fragmentation issues over time, since 823 V2 chunks are interleaved together with 734 V1 chunks. We have the same observations on other chunks and workloads, as shown in Fig.3(b), Fig.3(c), and Fig.3(e). The observation on MacOS is a little different, as shown in Fig.3(d). For example, the V1 chunks not only decrease in the second backup version, but also decrease in the third backup version. However, V1 chunks hardly decrease after these subsequent two backup versions are processed.

From the experimental results in Fig.3, we have two important observations. First, the adjacent backup versions are the most similar. Second, the chunks not appearing in the current backup version have a low probability to appear in subsequent backup ver-

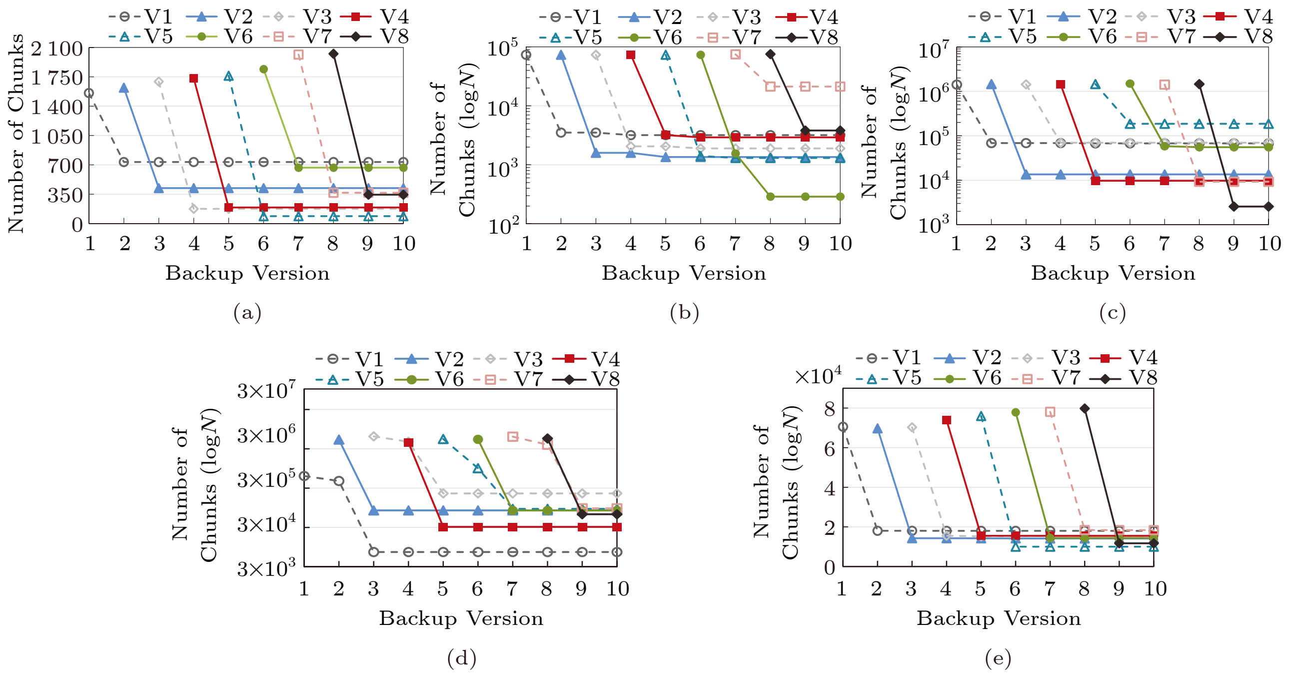

Fig.3. Chunk distributions of different workloads. (a) Linux Kernel. (b) GCC. (c) Fslhomes. (d) MacOS. (e) Boost.

sions[24]. The real-world applications also offer insights to prove these observations[20, 22, 23], e.g., a new version of software is upgraded from the old versions. The new version contains most contents of the old versions for software compatibility, and develops some new functions for better usage. Moreover, the system snapshots are generated along with time, and a new snapshot is generated from the old ones.

The obtained observations motivate us to store the chunks of new backup versions closely to enhance the physical locality for high restore performance, e.g., all the V8 chunks are stored closely to improve the physical locality of version 8. Although the restore performance of the old version decreases, such design is feasible since existing studies[20, 22, 23] demonstrate that the newer backup versions are more likely to be restored from the system crashes or version rollbacks than the older backup versions. It is worth noting that all the observations come from backup systems, e.g., the systems store different versions of the software (such as GCC, Linux Kernel) and the snapshots. We have the same observations on other workloads, e.g., Gdb and Cmake[4, 15, 30].

4 Design of HiDeStore

Unlike existing schemes, we propose HiDeStore to efficiently store chunks with high physical locality for high deduplication and restore performance. The workflow of our design is viewed as a reverse inline deduplication system. One of the key insights is to classify the hot and cold chunks during the deduplication phase. The chunks having a high probability to appear in new backup versions are hot chunks, while the other chunks become cold chunks. Another insight is to store the hot and cold chunks into the active and archival containers to enhance the physical locality, respectively. By grouping the chunks of new backup versions closely, the chunk fragmentation problem is alleviated and the restore performance is improved. Moreover, we directly remove the expired containers without expensive garbage collection.

The system overview of HiDeStore is shown in Fig.4. The differences with existing schemes are that HiDeStore identifies hot and cold chunks in the proposed fingerprint cache with double hashes, and stores chunks via a filter to gather different chunks in different containers. Specifically, the fingerprint cache identifies duplicate chunks when the coming chunks are matched within the fingerprint cache. The chunks not appearing in the current backup version become cold and are removed from the fingerprint cache after current backup version is processed. To improve the physical locality, HiDeStore temporarily stores the coming hot chunks in active containers and moves cold chunks to archival containers. In the context of our paper, the active and archival containers are stored in different locations to enhance the physical locality for hot and cold chunks, respectively. After the cold chunks are kicked out from the active containers, HiDeStore merges the sparse active contain-

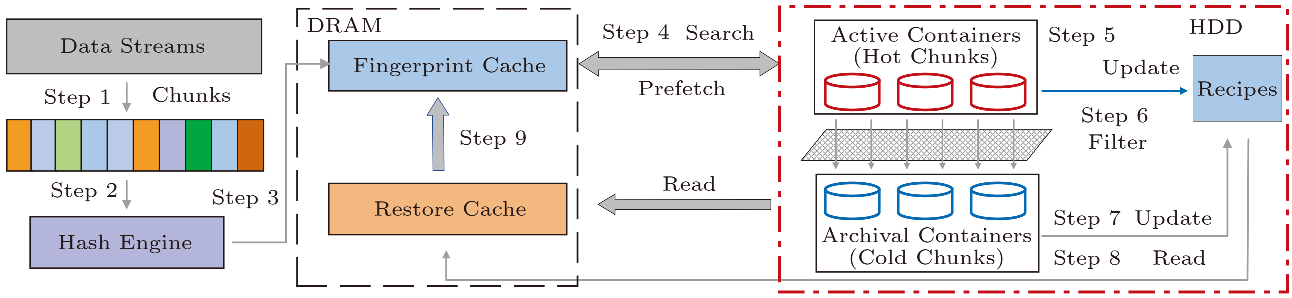

Fig.4. System overview of HiDeStore.

ers to improve the storage efficiency, and such design incurs acceptable overheads since the step of merging space containers is carried out offline. More details are shown in Subsections 4.1 and 4.2.

Moreover, the recipe records the locations of chunks when coming chunks are stored in different active containers, and it needs to be updated when some chunks are moved into archival containers. However, some chunks appear in multiple backup versions, incurring high overheads to update all the involved recipes. Instead of updating all recipes, HiDeStore proposes a recipe chain updating algorithm to only update the recipe of the previous one backup version, and a recipe chain is generated among multiple backup versions. To reduce the overheads of reading recipes, HiDeStore periodically eliminates the dependency of the recipe chain by pointing chunk references to the archival containers, as shown in Subsection 4.3. The original data streams are restored by reading chunks according to the recipes, and the workflow of restoring is shown in Subsection 4.4. Moreover, it becomes easy for HiDeStore to remove expired backups via the proposed ICDA, since the corresponding cold chunks are stored together in archival containers, as shown in Subsection 4.5.

4.1 Fingerprint Cache with Double Hash

The traditional fingerprint cache becomes inefficient to exploit the observations from Fig.3, since the cache fails to identify the hot and cold chunks during the deduplication phase. Moreover, the traditional fingerprint cache prefetches chunks according to the logical locality, and becomes inefficient to provide sufficient space for hot chunks since the cold chunks are also prefetched in the cache.

The observations from Fig.3 indicate that cold chunks have a negligible probability to appear in subsequent backup versions. Hence, we only need to store hot chunks in the fingerprint cache. Unlike existing schemes, we propose a fingerprint cache with two hash tables to classify the hot and cold chunks. In the deduplication phase, HiDeStore only searches hot chunks in the fingerprint cache and overlooks cold chunks to avoid the expensive disk accesses. The two hash tables are represented as T1 and T2, respectively, each of which contains fingerprints as keys and metadata of chunks as values, where the metadata consists of the chunk size and the IDs of active containers being stored (abbreviated as CIDs). Before current backup version (represented as CV) is processed, T1 caches the hot chunks (i.e., the chunks of the previous backup version) and T2 is empty. During the deduplication phase, the identified unique chunks are directly inserted into T2, while the chunks hitting T1 are removed from T1 and inserted into T2. After CV is processed, the chunks remaining in T1 become cold chunks since these chunks do not appear in CV, while the chunks in T2 are hot chunks and used to deduplicate subsequent backup versions.

The workflow of the proposed double-hash fingerprint cache is illustrated in Fig.5, which totally contains three kinds of cases to process the coming chunks. In the first case, chunk A is identified as a unique chunk due to not hitting both T1 and T2. We insert the fingerprints of chunk A into T2 and store the content of chunk A into an active container, as shown in Subsection 4.2. In the second case, chunk B is identified as a duplicate chunk due to hitting T1. Chunk B is also classified as a hot chunk due to having a high probability to appear in subsequent backup versions. In this case, we move the fingerprints of chunk B from T1 to T2 to process the subsequent backup versions. It is worth noting that the content of chunk B has been stored in an active container, since chunk B is a duplicate chunk. In the third case, chunk C is also identified as a duplicate chunk due to hitting T2. The metadata and content of chunk C have been correctly stored in T2 and active containers. After CV is processed, the chunks in T1 become cold and their contents are moved from active containers to archival containers, as shown in Subsection

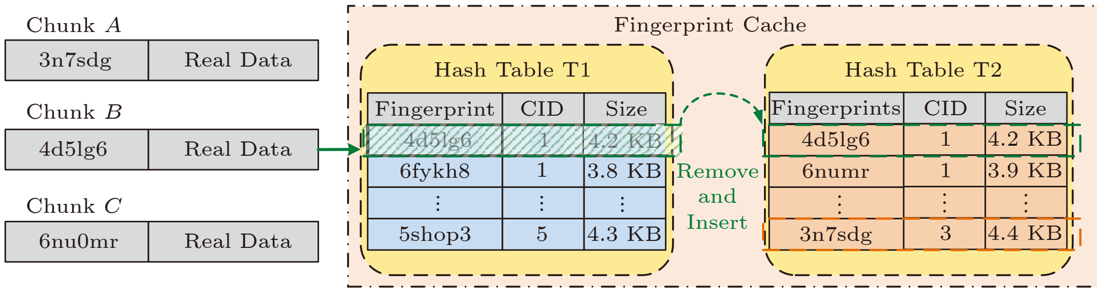

Fig.5. Structure of the fingerprint cache[24].

4.2. Finally, HiDeStore leverages the hot chunks to deduplicate the subsequent backup versions, which is simply implemented by changing T2 to T1.

We add another hash table to process the workload of MacOS, which is similar to the double-hash fingerprint cache to identify and classify hot and cold chunks. Since we use the fingerprints calculated via SHA-1 as keys in the hash table, the probability of a hash collision is much smaller than that of a hardware error[13]. It is worth noting that the sizes of T1 and T2 are bounded to the metadata size of one (or two) backup version(s), and hardly overflow the limited memory. Take the data in MacOS (a very large workload) as an example, i.e., one version contains about 5 million chunks and the total size of T2 is about 100 MB ($5\,000\,000\times28$ byte), where 28-byte metadata consists of 20-byte fingerprints, a 4-byte CID, and a 4-byte chunk size, as shown in Fig.5.

Compared with traditional deduplication schemes, HiDeStore significantly improves the deduplication throughput due to avoiding the expensive disk accesses. Moreover, HiDeStore achieves high deduplication ratios as shown in Subsection 5.2, since only hot chunks have a high probability to appear in the subsequent backup versions and searching hot chunks in the fingerprint cache is efficient for high deduplication ratios.

4.2 Chunk Filter to Separate Chunks

The traditional deduplication systems directly write the incoming unique chunks into containers for the archival purpose, which however incurs the severe chunk fragmentation problem as shown in Fig.2. Unlike existing schemes, HiDeStore changes the storage paths for the coming chunks, and stores hot and cold chunks into active and archival containers, respectively. The structures of active and archival containers are the same, as shown in Fig.6. A container contains the metadata and real data of chunks, where the metadata consists of the container ID, the total data size, and the hash table for the contained chunks. Each container has the same size with traditional containers (i.e., 4 MB) to achieve high storage efficiency.

Specifically, the incoming unique chunks are temporarily stored in active containers during the deduplication phase, served as hot chunks. After one backup version is deduplicated, the cold chunks are identified by the fingerprint cache and moved from active containers to archival containers. The process of chunk moving works like a filter, as shown in Fig.4. However, some active containers become sparse after the cold chunks are removed, and we need to compact the sparse containers to improve the space utilization. We cannot directly reuse the space of the removed chunks due to the unequal sizes. Specifically, the deduplication systems generally use content-based chunking algorithms to avoid the boundary-shift problem[9, 34], which generates variable-length chunks. For example, 7.3 KB space in total is released in container 1 after the chunks of 3.6 KB and 3.7 KB are removed, as shown in Fig.6. However, we cannot insert chunk E with 4.2 KB into container 1 due to the discontinuous space. Although chunk F with 3.1 KB

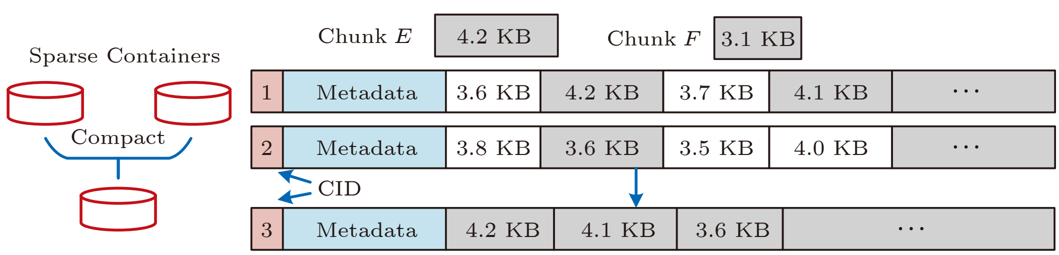

Fig.6. Compaction of sparse containers.

can be inserted into container 1, a large amount of fragmented space is generated and wasted.

Instead, HiDeStore merges and compacts the sparse containers to reuse the fragmented space in active containers. Specifically, HiDeStore calculates the space utilization for the active containers after the cold chunks are removed, where the space utilization is defined as the used size divided by the total size. The container with a low space utilization is identified as a sparse container, requiring to be merged. The process of compacting sparse containers is illustrated in Fig.6, which writes the chunks of two (or more) sparse containers into the same container without considering the order, since all these chunks are hot chunks and prefetched together during the reading phase. The merged container is stored on disk by overwriting the sparse container whose CID is the smallest to improve the space utilization.

To avoid the overheads of moving cold chunks from active containers to archival containers, HiDeStore implements the chunk moving phase in a pipeline manner with high parallelism. The deduplication system continues to process the next backup version without waiting for moving chunks, since the hot and cold chunks have been identified in the proposed double-hash fingerprint cache and the cold chunks are moved to archival containers offline. By separately storing different chunks in active and archival containers, HiDeStore improves the physical locality of new backup versions. HiDeStore achieves higher restore performance due to incurring fewer expensive disk accesses compared with existing schemes[15–18].

4.3 Recipes Updating

The deduplication systems record all chunk references of the original data stream in the recipes for future restoring, where a chunk reference consists of the fingerprints, the chunk size, and the ID of the corresponding container (represented as CID). In HiDeStore, the chunks are temporarily stored in active containers and moved to archival containers when the chunks become cold. To exactly record the locations for chunks, the recipes need to be updated when the chunks are moved to different containers. However, we have to check all recipes to determine which one needs to be updated, which incurs high overheads due to the expensive disk accesses.

We propose a recipe chain updating algorithm to reduce the overheads of updating recipes, which only updates the previous one recipe, rather than checking all recipes. For the case of MacOS, we update the previous two recipes. The recipe chain updating algorithm is illustrated in Fig.7, which shows the results after backup version V_4 is processed. Since V_4 is the newest backup version and all chunks are hot chunks, the recipe R_4 of V_4 records CIDs of all chunks as 0, indicating that all chunks are stored in active containers. HiDeStore obtains the specific active container by checking the fingerprint cache. At the same time, we update R_3 since some chunks of V_3 become cold and are moved to archival containers, which is implemented by modifying CIDs of these chunks to the IDs of the corresponding archival containers. The CIDs of remaining chunks in R_3 are modified to the negative ID of V_4, e.g., the CID -4 indicates that we need to further check R_4 to find the final chunk locations, while the CID 4 indicates that the chunk is stored in the archival container 4.

As a result, all recipes form a chain as shown in Fig.7. However, determining the locations of chunks incurs high overheads due to the needs of checking multiple recipes in the recipe chain. To eliminate the dependency among recipes, HiDeStore periodically updates the chunk references via Algorithm 1, which updates recipes from back to front. We use N to represent the newest backup version, while using n to represent the previous backup version. Algorithm 1 updates the recipes from R_{N-1}, since the newest recipe R_N stores all chunks in the active containers and does not need to read another recipe to determine the chunk locations. Specifically, HiDeStore reads the recipe R_{N-1} and inserts all positive CIDs into the hash table T (lines 1–6), indicating that the corresponding chunks are stored in archival containers.

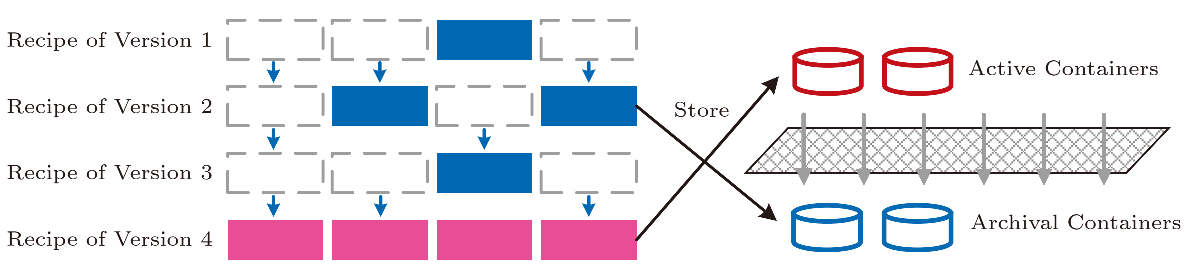

Fig.7. Recipes updating. The blue and red chunks are stored in archival and active containers, respectively.

Hash table T is used to update the previous recipe R_{N-2}, which modifies the negative CID of a chunk in R_{N-2} to the positive CID in T when the chunk has a match in T (lines 11 and 12). The remaining negative CIDs are modified to $-(n+1)$ (lines 13 and 14), indicating that the chunk references are obtained from the next recipe. At the same time, HiDeStore inserts the positive CIDs of R_{N-2} into a new hash table T' (lines 17–19) to update the previous recipe R_{N-3} (line 22). Finally, all recipes point to R_N to obtain the locations of chunks. It is worth noting that $-N$ in the recipe indicates that the chunks are stored in active containers. Moreover, HiDeStore updates recipes from R_N next time, rather than reading all the recipes to eliminate the recipe chain.

Algorithm 1. Recipes Updating[24]

Input: recipe $R[N]$, hash table T
Output: updated recipe $R[N]$
1: int $n = N - 1$; // n is the previous backup version
2: **for all** chunk c in $R[n]$ **do**
3: **if** $c.CID > 0$ **then**
4: insert chunk c into hash table T
5: **end if**
6: **end for**
7: $n - -$; // update the recipes of older backup versions
8: **while** recipe $R[n]$ exist **do**
9: **for all** chunk c in $R[n]$ **do**
10: **if** $c.CID < 0$ **then**
11: **if** chunk c matches chunk p in T **then**
12: $c.CID = p.CID$
13: **else**
14: $c.CID = -(n+1)$
15: **end if**
16: **end if**
17: **if** $c.CID > 0$ **then**
18: insert chunk c into a new Hash Table T'
19: **end if**
20: **end for**
21: $HashTableDestroy(T)$ and $T = T'$;
22: $n - -$; // update the recipes of older backup versions
23: **end while**

Updating recipes incurs negligible overheads due to the small sizes of the recipe files. The recipes only record the metadata of chunks, as shown in Subsection 5.4. Moreover, HiDeStore updates the recipes offline to avoid blocking the deduplication system.

4.4 Restore Phase

Original data are restored according to the chunk references in recipes. In traditional deduplication systems, all CIDs in recipes are positive numbers and indicate the referred containers. However, HiDeStore contains three types of CIDs in recipes, including positive CIDs, 0, and negative CIDs. The positive CIDs and negative CIDs indicate the archival containers and the backup versions, respectively; while 0 indicates the active containers. In this case, we update recipes according to Algorithm 1 to obtain the locations for all chunks, and then read the contents of chunks from the active and archival containers.

The obtained chunks assemble the original data stream in the restore cache via the chunk- and container-based approaches[15–18, 20]. Compared with existing schemes, HiDeStore enhances the physical locality for the new backup versions and delivers higher restore throughput, since fewer disk accesses are incurred for chunk reading.

4.5 Removing of Expired Versions

In deduplication systems, expired versions are removed for saving space[15, 23]. However, we cannot 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 the chunks are removed, which however incurs high overheads due to the needs of checking all backup versions. Moreover, the chunks of different versions are interleaved together, as shown in Fig.2, requiring some garbage collection efforts to reclaim the space for the deleted chunks. The challenge is to remove the chunks that are only referred by expired versions while not incurring a large number of efforts for garbage collection. In practice, HiDeStore is efficient to carry out chunk detection and garbage collection, since the chunks of different backup versions are stored in different containers. Unlike existing schemes that count the references of chunks, HiDeStore leverages the Isolated Container Deletion Algorithm (ICDA) to remove the containers only referred by expired backup versions.

The methodology of ICDA is based on the classification of hot and cold chunks. The different chunks are identified via the proposed fingerprint cache and stored in active and archival, containers respectively. The cold chunks of the previous backup versions are not referred by the subsequent backup versions according to the observation from Fig.3. Instead, the cold chunks are physically gathered in the same archival containers. As shown in Fig.8, we temporarily store hot chunks in active containers when different backup versions are processed. At the beginning,

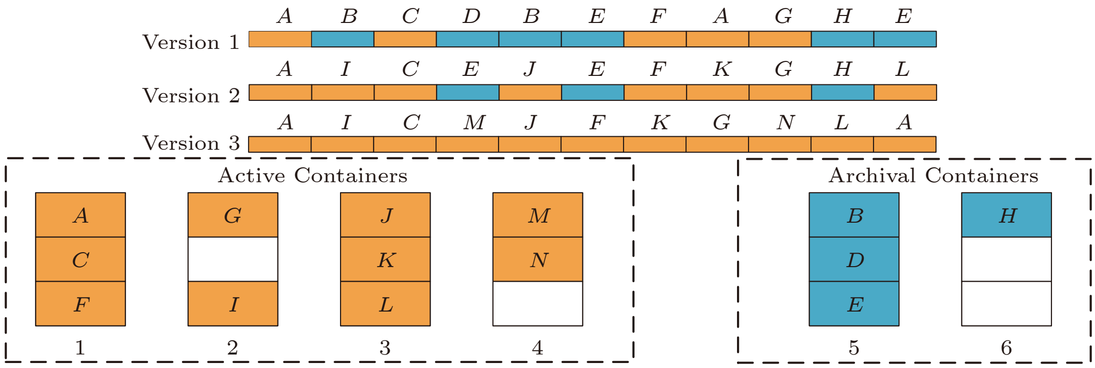

Fig.8. Removing of expired data.

all chunks of backup version 1 are stored in active containers, since these chunks have a high probability to appear in subsequent backup versions. When we process the backup versions 2 and 3, some hot chunks become cold, e.g., chunks B, D, E, and H. In this case, we move these cold chunks from active containers to the archival containers. Obviously, the cold chunks (e.g., chunks B, D, E, and H) are not referred by version 3. We directly remove the archival containers 5 and 6 when the expired versions 1 and 2 are deleted, avoiding the expensive expired data detection and garbage collection, due to the high physically locality of the expired data.

However, in the case where only the backup version 1 is deleted, we cannot directly remove the archival container 5, since the chunk E is also referred by the non-expired backup version 2. To efficiently detect the archival containers that are only referred by the expired backup versions, ICDA maintains extra 8 B metadata in the container to record the ID of the newest backup version (represented as $B_{\rm ID}$). The container whose $B_{\rm ID}$ is not larger than that of the expired version is the expired container. We directly remove these expired containers without the needs for expensive chunk detection and garbage collection, since these containers are not referred by the non-expired backup versions.

Unlike existing schemes that detect expired backup versions chunk by chunk, ICDA removes the expired data in the granularity of containers. By physically grouping the expired chunks together in the archival containers, ICDA becomes efficient to remove the expired data and reclaim continuous storage space for further usage.

5 Performance Evaluation

We compare HiDeStore with state-of-the-art schemes in terms of deduplication and restore performance.

5.1 Experimental Setup

The prototype of HiDeStore is implemented based on a widely used deduplication framework, called Destor[4], which processes data in a pipeline with high parallelism. Unlike traditional deduplication schemes, HiDeStore modifies the indexing, rewriting, and storing phases to identify and classify the hot and cold chunks. To facilitate fair comparisons, HiDeStore uses a TTTD (Two-Threshold Two-Divisor) chunking algorithm[27] and SHA-1 hash functions like other schemes in the chunking and hashing phases to generate fingerprints for further deduplicaiton. Moreover, HiDeStore stores the fingerprints in the hash tables for low hash collisions[12].

To show the efficiency of HiDeStore in terms of deduplication performance, we select state-of-the-art locality- and similarity-based schemes for comparisons, including DDFS (Data Domain File System)[13], Sparse Index[14], and SiLo[30]. DDFS removes all duplicate chunks by searching the whole fingerprint table to achieve the highest deduplication ratio. Sparse Index samples parts of fingerprints for caching to reduce the overheads of searching the whole fingerprint table, which significantly reduces the memory consumption for the fingerprint cache. By exploiting the similarity of chunk streams, SiLo further improves the throughput for deduplication. Moreover, to show the efficiency of HiDeStore in terms of restore performance, we compare state-of-the-art caching- and rewriting-based schemes, including Capping[17], ALACC (Adaptive Look-Ahead Chunk Caching)[18], and LBW (Low-Back Window)[21]. We directly run the source codes of ALACC for evaluations, while re-implementing FBW according to the original work[21] due to the lack of the open source codes. We configure all schemes with the reported parameters from the original work to achieve the best results.

We conduct experimental evaluations on five widely used datasets[4, 15, 18, 21, 30], and the details of these datasets are shown in Table 1. We conduct all experiments on a Linux server with kernel version v4.4.114. The server is equipped with two 8-core Intel® Xeon® E5-2620 v4 @2.10 GHz CPUs (each core with 32 KB L1 instruction cache, 32 KB L1 data cache, and 256 KB L2 cache), 20 MB last level cache and 24 GB DRAM.

Table 1. Details of Datasets

Dataset	Total Size	Total Versions	Dedup_Ratio (%)
Linux Kernel②	64.0 GB	158	91.53
GCC③	105.0 GB	175	78.75
Boost④	61.0 GB	38	83.42
Fslhomes⑤	920.0 GB	102	92.17
MacOS⑤	1.2 TB	25	89.56

5.2 Performance in Deduplication Phase

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

5.2.1 *Deduplication Ratio*

The deduplication ratio examines the amount of data reduced by the deduplication system, which is calculated via dividing the size of eliminated data by the total data size. The deduplication ratios of different deduplication schemes are shown in Fig.9. From the results, we observe that the deduplication ratio of DDFS is the highest, since DDFS removes all identified duplicate data by searching the whole fingerprint table. Unlike DDFS, Sparse Index and SiLo only search the cached fingerprints in memory to reduce the overheads of frequent disk accessing, which however decreases some deduplication ratios since some duplicate chunks are overlooked. Specifically, Sparse Index and SiLo group multiple chunks into segments and sample partial chunks as features. Two segments sharing the same features are identified as similar segments. Sparse Index and SiLo only search the similar segments to identify the duplicate chunks. However,

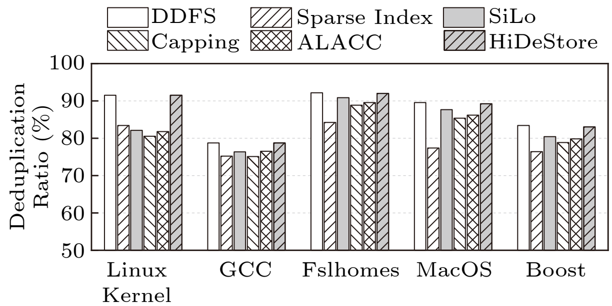

Fig.9. Deduplication ratios.

some duplicate chunks in the segments are not sampled as features. As a result, these duplicate chunks are not searched during the deduplication phase and result in low deduplication ratios. Although configuring a large sampling ratio achieves a high deduplication ratio, more memory is consumed for fingerprint caching and longer latency is incurred for features searching. Moreover, we observe that the deduplication ratio of HiDeStore is almost the same with that of DDFS, since HiDeStore caches the chunks that have high probabilities to be deduplicated. These chunks are identified via our proposed double-hash fingerprint cache, which fully exploits the observations from Fig.3, i.e., only the hot chunks appear in the subsequent backup versions. By searching the hot chunks, HiDeStore efficiently identifies the duplicate chunks.

Moreover, we also evaluate the deduplication ratios for the rewriting schemes, and the results are shown in Fig.9. We observe that the deduplication ratios of the rewriting schemes are lower than those of the other schemes, since the stored duplicate chunks occupy the available storage space and decrease the storage efficiency. Moreover, the rewriting schemes further decrease the deduplication ratios when more data are processed due to the existence of more duplicate chunks.

5.2.2 *Deduplication Throughput*

The experimental platform, i.e., Destor[4], evaluates the number of the lookup requests to the disk to show the overheads of the deduplication phase. Specifically, Destor maintains the whole fingerprint table on disk for fingerprint searching, while caching parts of fingerprints in memory to accelerate the fingerprint searching phase. Hence, a large number of

②Linux Kernel, https://www.kernel.org, Nov. 2024.
③GCC, https://ftp.gnu.org/gnu/gcc, Nov. 2024.
④Boost, https://www.boost.org, Nov. 2024.
⑤Snapshots, https://tracer.filesystems.org, Nov. 2024.

the lookup requests for the whole fingerprint table represent the high overhead of accessing disk, which delivers low deduplication throughput due to the expensive disk I/Os. We evaluate the lookup requests per GB like Destor to show the deduplication throughput of different schemes, where the lookup requests per GB are defined as the number of lookup requests for the whole fingerprint table when 1 GB data are processed. Unlike conventional schemes, HiDeStore identifies and classifies the hot and cold chunks during the deduplication phase. All the hot chunks are prefetched in the fingerprint cache before the deduplication phase begins, and HiDeStore only searches the cached hot chunks to avoid the high overheads of frequent disk accessing. We calculate the lookup requests of HiDeStore with the same unit size as the conventional schemes to facilitate fair comparisons, and the results are shown in Fig.10.

From the results, we observe that the lookup requests of HiDeStore are the lowest among all schemes. Specifically, HiDeStore reduces the lookup overheads by up to 140%, 50%, and 24% than DDFS, Sparse Index, and SiLo, respectively. That is because HiDeStore only searches the hot chunks in the fingerprint cache, and the duplicate chunks have a high probability to match with these hot chunks according to the observations from Fig.3. By avoiding frequently accessing the whole fingerprint table on disk, HiDeStore reduces the overheads of fingerprint searching and achieves high deduplication throughput.

From the results in Fig.10(d), we observe that HiDeStore incurs higher lookup overhead than SiLo on MacOS, because HiDeStore prefetches the chunks of the last two backup versions in the fingerprint cache. However, it is worth noting that the hot chunks of the last two versions are prefetched in the fingerprint cache before the next backup version is processed. The lookup overheads on MacOS incurred by HiDeStore are negligible, since the prefetching of HiDeStore does not block the deduplication phase. Moreover, HiDeStore sequentially prefetches fingerprints from the recipe, and is more efficient than traditional deduplication schemes due to the efficient sequential read performance.

5.2.3 *Space Consumption for Fingerprint Table*

The deduplication system stores the fingerprints in a hash table for further deduplication, which identifies and removes duplicate chunks when the fingerprint table has a match with the coming chunk. The traditional deduplication schemes maintain all or sample parts of fingerprints in the fingerprint table, depending on the sampling ratios. Unlike the traditional deduplication schemes, HiDeStore directly reads hot chunks from the recipe of the previous backup version, avoiding constructing an extra fingerprint table to store the metadata, and hence showing significant strengths over existing schemes. We use the same metric as existing schemes[4, 30], i.e., space overhead per MB/B[30], to evaluate the space consumption for the fingerprint table, where the space overhead per MB is defined as the required space for

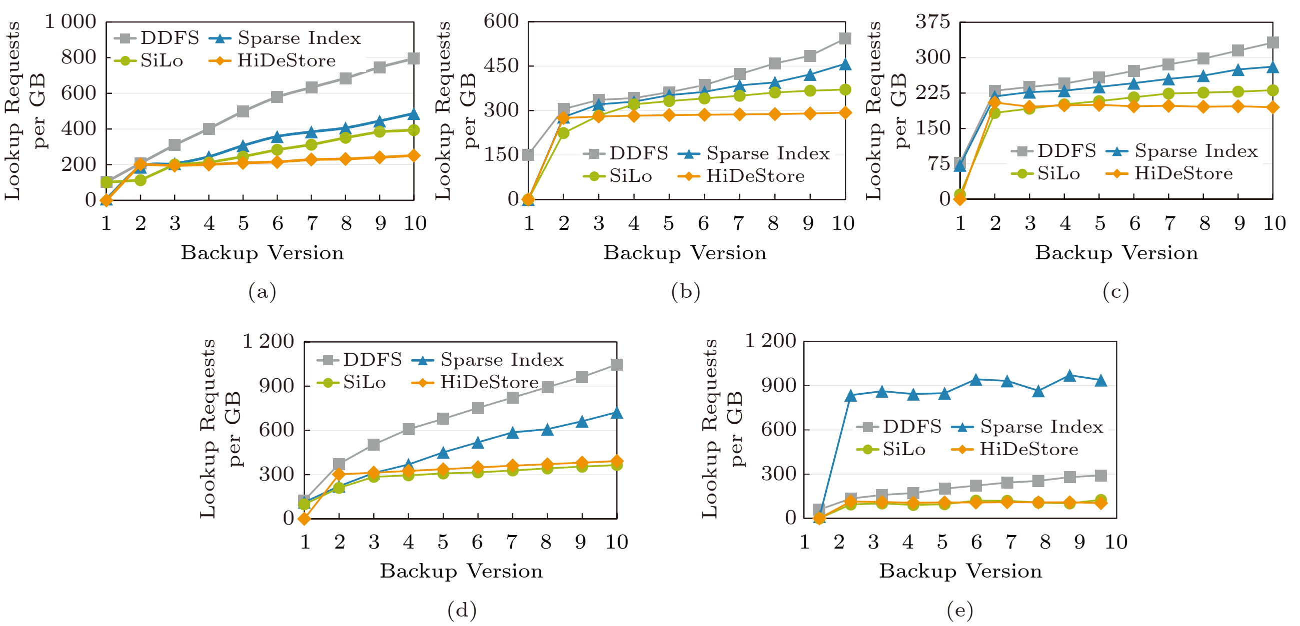

Fig.10. Lookup overheads on different workloads. (a) Linux Kernel. (b) GCC. (c) Fslhomes. (d) MacOS. (e) Boost.

the indexes to deduplicate 1 MB data.

We evaluate the space overhead per MB/B for the fingerprint tables of all schemes, and the results are shown in Fig.11. DDFS incurs the highest space consumption for the fingerprint table, since DDFS stores all fingerprints of unique chunks for exact deduplication. The space consumption is high when a large number of small files exist in the processed dataset, since a large number of chunks are generated. To reduce the space consumption of the fingerprint table, Sparse Index and SiLo leverage different sampling approaches and ratios to maintain parts of the fingerprints for near-exact deduplication, and outperform DDFS by up to two orders-of-magnitude. For example, Sparse Index achieves about 128x space savings when the sample ratio is set to 128 : 1. SiLo further reduces the space consumption for the fingerprint table, since SiLo samples less fingerprints from a segments than Sparse Index.

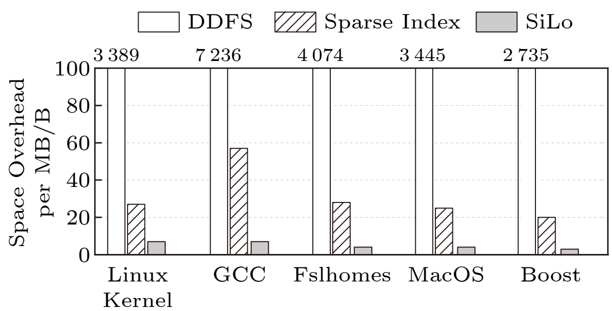

Fig.11. Index table overheads.

Unlike the traditional schemes, HiDeStore does not require extra space to store the fingerprint table, due to identifying and maintaining the fingerprints of hot chunks during the deduplication phase. Specifically, the hot fingerprints have been stored in the recipe of the previous backup version. The hot fingerprints are directly prefetched in the fingerprint cache before the next backup version is processed. Therefore, HiDeStore has significant strengths over existing deduplication schemes in terms of the space consumption. Moreover, HiDeStore saves more storage space than existing schemes when more backup versions are processed, since HiDeStore does not need extra space for the fingerprint table while existing schemes proportionally consume a large amount of storage space to store the fingerprint tables.

5.3 Performance in Restore Phase

The restore phase assembles the original data in a chunk-by-chunk manner, requiring to read chunks from various containers on disks according to the recipe. The speed of restoring data is significantly influenced by the performance of reading chunks. Existing schemes deliver low restore performance due to incurring a large number of disk I/Os to read the physically scattered chunks. The restore performance decreases when the backup system stores multiple backup versions due to the severe chunk fragmentation problem. Unlike existing schemes, HiDeStore aims to achieve high restore performance by enhancing the physical locality of the data. We use the same metric with existing schemes[15–18, 21] to evaluate the restore performance, i.e., a speed factor (MB/container-read) which is defined as the mean data size that is restored per container[17, 21]. The biggest advantage of the speed factor is to avoid the speed variances of different data servers. The low speed factor indicates that the chunks are physically scattered into different containers, which delivers low restore performance due to the chunk fragmentation problem. We set the sizes of all containers to 4 MB to facilitate fair comparisons. The scheme without the rewriting phase is set to be the baseline. Moreover, we also compare HiDeStore with state-of-the-art rewriting schemes to show the efficiency of HiDeStore over existing schemes.

Fig.12 shows the restore performance of different schemes. We observe that existing schemes deliver high restore performance on the old backup versions, while delivering low restore performance on the new backup versions, because the chunk fragmentation problem is exacerbated over time, as shown in Fig.2. Unlike existing schemes, HiDeStore significantly improves the restore performance for the new backup versions, e.g., the restore performance of HiDeStore is about 2.6x higher than that of LBW+ALACC on the new backup versions. The main reason is that the physical locality of the new backup versions is enhanced by the proposed active and archival containers. Specifically, HiDeStore temporarily maintains hot chunks in active containers. When some hot chunks become cold in processing the subsequent versions, HiDeStore moves these chunks to archival containers. Through this way, the hot chunks of new backup versions are stored closely to avoid the chunk fragmentation problem, and the restore performance of the new backup version is significantly improved. It is worth noting that the new backup versions are more likely to be restored than the old backup versions[20, 22, 23] for version rollbacks, and HiDeStore is efficient to

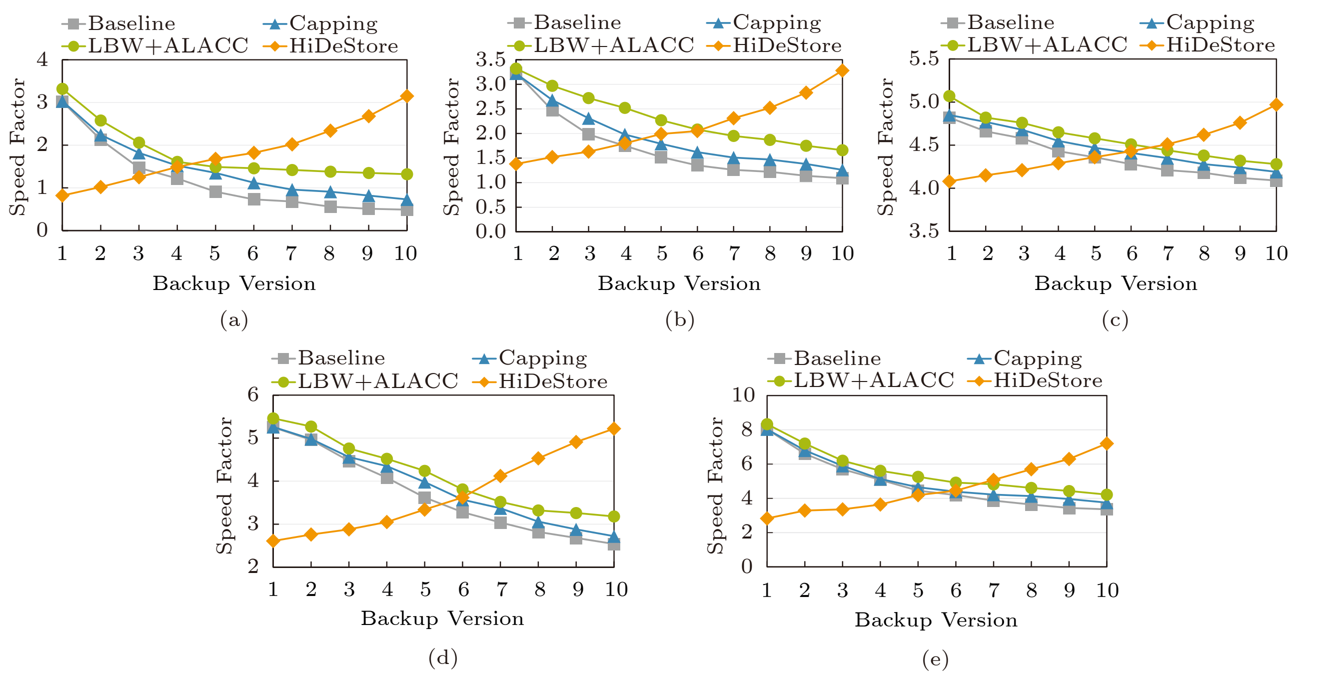

Fig.12. Restore performance on different workloads. (a) Linux Kernel. (b) GCC. (c) Fslhomes. (d) MacOS. (e) Boost.

meet the demands of restoring the new backup versions with high performance. Moreover, compared with the rewriting schemes, we observe that HiDeStore not only delivers higher restore performance on the new backup version, but also achieves higher deduplication ratios, as shown in Fig.9 and Fig.12. The main reason is that HiDeStore physically stores the hot chunks together in the same containers, rather than rewriting multiple duplicate chunks to consume a large amount of storage space.

5.4 Overheads Incurred by HiDeStore

We evaluate the overheads incurred by HiDeStore, including the time overheads of updating recipes and moving chunks. Specifically, HiDeStore records the locations for the stored hot chunks during the deduplication phase. When some hot chunks become cold after one backup version is processed, HiDeStore moves these chunks from active containers to archival containers, and updates the recipe according to Algorithm 1 for future restoring. Algorithm 1 incurs $O(N)$ complexity, where N is the number of recipes. We evaluate the latency of updating a recipe on different datasets, and the results are shown in Fig.13. We observe that the updating latency is related to the size of a dataset, e.g., HiDeStore spends 21 ms on updating a recipe for the dataset of Linux Kernel. Moreover, it is worth noting that HiDeStore updates the recipes after a backup version is processed, which does not block the deduplication system.

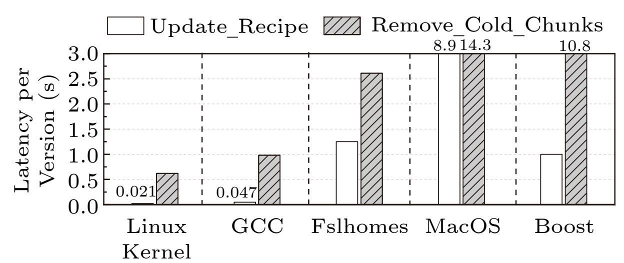

Fig.13. Overheads incurred by HiDeStore, including the overheads of moving cold chunks and updating recipes.

The overheads of moving chunks from active containers to archival containers are higher than those of the recipe updating phase, as shown in Fig.13. However, the chunk moving phase is implemented in a pipeline manner with high parallelism based on Destor, avoiding blocking the deduplication system for a long time. Moreover, HiDeStore moves chunks and merges sparse containers offline to avoid the long latency penalty, and hence the overheads of moving chunks are acceptable in HiDeStore.

5.5 Expired Backup Deletion

The expired backup versions are removed to save space[15, 23], which needs to detect the expired chunks, i.e., the chunks are only referred by the expired backup versions. We evaluate the metadata overheads for different inline reference management schemes, including Reference Counter (RC)[32], Grouped Mark-and-Sweep (GMS)[33], the Container-Marker Algorithm (CMA)[15], and our proposed Isolated Container Deletion Algorithm (ICDA). We use one byte to maintain the reference counter in different schemes for fair comparisons, and the results are shown in Fig.14. RC causes the highest metadata overhead due to recording references for all unique chunks. GMS maintains a bitmap in the container for references, and incurs high metadata overhead since each container stores a large number of chunks. CMA and ICDA record references for containers, and each container only consumes one byte for the counter. Therefore, CMA and ICDA achieve about three orders of magnitude space savings than previous schemes. Moreover, ICDA saves more space than CMA, since ICDA does not record references for active containers.

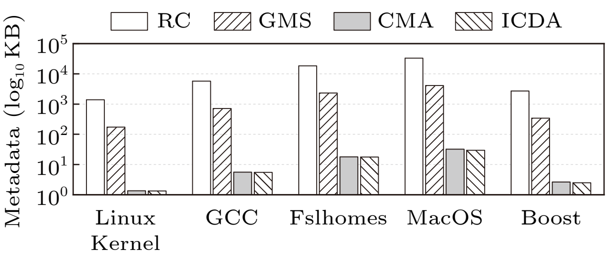

Fig.14. Reference metadata overheads of different schemes.

Garbage collection needs to merge sparse containers after the chunks are removed. Compared with RC and GMS, CMA and ICDA generate fewer sparse containers and significantly reduce the garbage collection overheads, since CMA and ICDA directly remove the expired containers rather than the scattered chunks. We examine the number of non-expired containers after the expired backups are removed to exhibit the actual storage cost, i.e., using the same metric with CMA[15] to facilitate fair comparisons. Apart from the basic deduplication process (represented as the baseline), we also evaluate the impact of rewriting algorithms, and the results are shown in Fig.15. We observe that ICDA shows large advantages over other schemes, and reduces the numbers of containers by up to 1.8x, 2.0x, and 1.5x than the baseline, Capping, and CMA schemes, respectively. The main reason is that the cold chunks of a backup version are gathered together in the same archival containers, and these archival containers are directly removed during the expired backup deletions. However, other schemes store the cold chunks in multiple containers and fail to fully utilize the containers after the expired chunks are removed. Moreover, the rewriting algorithm consumes more space than the other schemes even after the expired backups are removed, since multiple chunks for the new backup versions are rewritten.

6 Related Work

Deduplication Schemes for Fingerprint Access

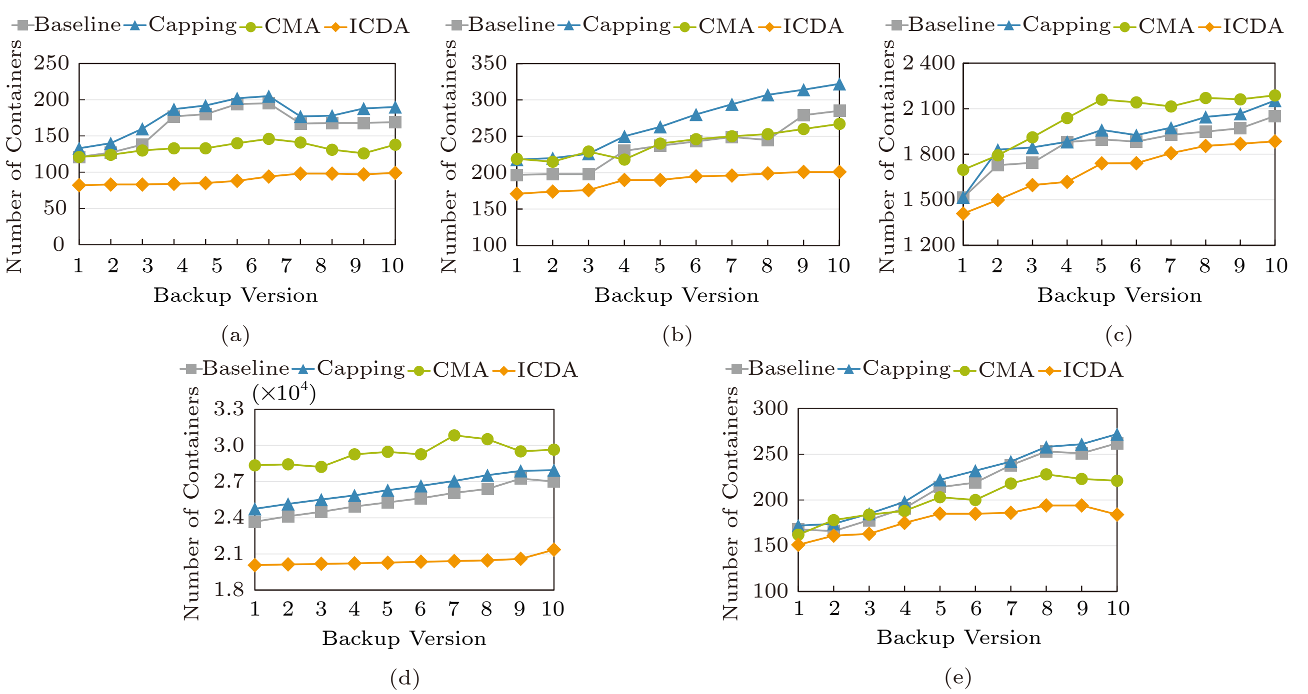

Fig.15. Number of non-expired containers after the expired backup versions are removed. (a) Linux Kernel. (b) GCC. (c) Fslhomes. (d) MacOS. (e) Boost.

Bottleneck. The deduplication system stores fingerprints of all chunks on disk[13, 14]. However, the deduplication throughput significantly decreases when a large number of chunks are stored due to the expensive disk I/Os. Zhu *et al.*[13] observed that the chunk sequences appear in the same order in multiple backup streams. By exploiting such logical locality of the chunk sequence, DDFS[13] proposes to prefetch the chunk sequences for caching and construct an in-memory bloom filter to deliver high deduplication throughput. To reduce the memory consumption of the cached chunks, Sparse Index[14] proposes to sample parts of chunks for near-exact deduplication. Block Locality Cache (BLC)[26] proposes to update the locality information according to the stored data, which avoids using the outdated locality. ChunkStash[25] stores the fingerprint table on SSD to avoid the penalty of the random disk I/O. Extreme Binning[29] and SiLo[30, 35] explore and exploit the similarity of segments to achieve high deduplication ratios.

Restore Schemes for Chunk Fragmentation Problem. Existing deduplication systems propose two kinds of approaches to alleviate the chunk fragmentation problem, including optimizing the restore cache and rewriting some duplicate chunks. Specifically, the stored chunk sequences have a high probability to be read for assembling the original data[17, 18]. Therefore, many schemes propose to cache the chunks and containers[15, 16, 20] to reduce the number of disk I/Os. ALACC[18] proposes to construct a cache for the sliding window to deliver higher restore performance. Unlike the caching-based schemes, many schemes rewrite chunks according to different standards, such as the Content-Based Rewriting algorithm (CBR)[16], Chunk Fragmentation Level (CFL)[22], and Capping[17]. Cao *et al.*[21] dynamically set the threshold for capping-based schemes on different workloads to enhance the physical locality of data streams.

Expired Data Deletion. The backup systems remove the expired data to save space. However, the expired chunks are physically scattered into different containers and become hard to be removed due to the high overheads of expired chunk detection and garbage collection. Reference Counter (RC)[32] counts the reference number of chunks, and removes the chunks which are not referred by any backup version. To reduce the space overheads of referencing, Grouped Mark-and-Sweep (GMS)[33] uses a bitmap in each container, while the Container-Marker Algorithm (CMA)[15] marks the containers rather than chunks.

7 Conclusions

Based on the observation that the adjacent versions are the most similar, our proposed HiDeStore leverages the double-hash fingerprint cache to identify hot and cold chunks, and stores different chunks in active and archival containers respectively to enhance the physical locality. Our experimental evaluation results show that HiDeStore achieves higher performance in terms of deduplication, restore, and data deletion than state-of-the-art schemes. We have released the open source code of HiDeStore for public use in GitHub[⑥]. Moreover, apart from the backup storage systems, the database systems and cloud storage systems contain a large amount of redundant data and require efficient deduplication techniques to save space. However, we cannot directly deploy HiDeStore in databases and cloud storage systems, since the data in these two systems exhibit different patterns. We will further optimize HiDeStore by exploring and exploiting the data patterns in other storage systems for better performance.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Khorasani S O, Rellermeyer J S, Epema D. Self-adaptive executors for big data processing. In *Proc. the 20th International Middleware Conference*, Dec. 2019, pp.176–188. DOI: 10.1145/3361525.3361545.

[2] Birke R, Rocha I, Perez J, Schiavoni V, Felber P, Chen L Y. Differential approximation and sprinting for multi-priority big data engines. In *Proc. the 20th International Middleware Conference*, Dec. 2019, pp.202–214. DOI: 10.1145/3361525.3361547.

[3] Akbari A, Martinez J, Jafari R. Facilitating human activity data annotation via context-aware change detection on smartwatches. *ACM Trans. Embedded Computing Systems*, 2021, 20(2): 15. DOI: 10.1145/3431503.

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

[5] Li Y K, Xu M, Ng C H, Lee P P C. Efficient hybrid in-

⑥The source code of HiDeStore is available at https://github.com/iotlpf/HiDeStore, Nov. 2024.

line and out-of-line deduplication for backup storage. *ACM Trans. Storage*, 2015, 11(1): Article No. 2. DOI: 10.1145/2641572.

[6] Park D, Fan Z, Nam Y J, Du D H C. A lookahead read cache: Improving read performance for deduplication backup storage. *Journal of Computer Science and Technology*, 2017, 32(1): 26–40. DOI: 10.1007/s11390-017-1680-8.

[7] Duggal A, Jenkins F, Shilane P, Chinthekindi R, Shah R, Kamat M. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *Proc. the 2019 USENIX Annual Technical Conference*, Jul. 2019, pp.647–660.

[8] Meyer D T, Bolosky W J. A study of practical deduplication. *ACM Trans. Storage*, 2012, 7(4): Article No. 14. DOI: 10.1145/2078861.2078864.

[9] Muthitacharoen A, Chen B, Mazières D. A low-bandwidth network file system. In *Proc. the 18th ACM Symposium on Operating Systems Principles*, Oct. 2001, pp.174–187. DOI: 10.1145/502034.502052.

[10] Wallace G, Douglis F, Qian H, 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*, Feb. 2012, p.4.

[11] Yang Q, Jin R, Zhao M. SmartDedup: Optimizing deduplication for resource-constrained devices. In *Proc. the 2019 USENIX Annual Technical Conference*, Jul. 2019, pp.633–646.

[12] Quinlan S, Dorward S. Venti: A new approach to archival storage. In *Proc. the FAST 2002 Conference on File and Storage Technologies*, Jan. 2002, pp.89–101.

[13] Zhu B, Li K, Patterson R H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. the 6th USENIX Conference on File and Storage Technologies*, Feb. 2008, pp.269–282.

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

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

[16] Kaczmarczyk M, Barczynski M, Kilian W, Dubnicki C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proc. the 5th Annual International Systems and Storage Conference*, Jun. 2012, Article No. 15. DOI: 10.1145/2367589.2367600.

[17] 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*, Feb. 2013, pp.183–198.

[18] Cao Z, Wen H, Wu F, Du D H C. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proc. the 16th USENIX Conference on File and Storage Technologies*, Feb. 2018, pp.309–324.

[19] Mao B, Jiang H, Wu S, Fu Y, Tian L. SAR: SSD assisted restore optimization for deduplication-based storage systems in the cloud. In *Proc. the 7th IEEE International Conference on Networking, Architecture*, Jun. 2012, pp.328–337. DOI: 10.1109/NAS.2012.48.

[20] Nam Y J, 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 Simulation of Computer and Telecommunication Systems*, Aug. 2012, pp.201–208. DOI: 10.1109/MASCOTS.2012.32.

[21] Cao Z, Liu S, Wu F, Wang G, Li B, Du D H C. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proc. the 17th USENIX Conference on File and Storage Technologies*, Feb. 2019, pp.129–142.

[22] Nam Y, Lu G, Park N, Xiao W, 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*, Sept. 2011, pp.581–586. DOI: 10.1109/HPCC.2011.82.

[23] Ng C H, Lee P P C. RevDedup: A reverse deduplication storage system optimized for reads to latest backups. In *Proc. the 4th Asia-Pacific Workshop on Systems*, Jul. 2013, Article No. 15. DOI: 10.1145/2500727.2500731.

[24] Li P, Hua Y, Cao Q, Zhang M. Improving the restore performance via physical-locality middleware for backup systems. In *Proc. the 21st International Middleware Conference*, Dec. 2020, pp.341–355. DOI: 10.1145/3423211.3425691.

[25] Debnath B K, Sengupta S, Li J. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. the 2010 USENIX Annual Technical Conference*, Jun. 2010, Article No. 16.

[26] Meister D, Kaiser J, Brinkmann A. Block locality caching for data deduplication. In *Proc. the 6th International Systems and Storage Conference*, Jul. 2013, Article No. 15. DOI: 10.1145/2485732.2485748.

[27] Eshghi K, Tang H K. A framework for analyzing and improving content-based chunking algorithms. Technical Report, HP Laboratory, 2005. http://shiftleft.com/mirrors/www.hpl.hp.com/techreports/2005/HPL-2005-30R1.pdf. Oct. 2024.

[28] Xia W, Zhou Y, Jiang H, Feng D, Hua Y, Hu Y, Liu Q, Zhang Y. Fastcdc: A fast and efficient content-defined chunking approach for data deduplication. In *Proc. the 2016 USENIX Annual Technical Conference*, Jun. 2016, pp.101–114.

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

10.1109/MASCOT.2009.5366623.

[30] Xia W, Jiang H, Feng D, Hua Y. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proc. the 2011 USENIX Conference on USENIX Annual Technical Conference*, Jun. 2011.

[31] Xu G, Tang B, Lu H, Yu Q, Sung C W. LIPA: A learning-based indexing and prefetching approach for data deduplication. In *Proc. the 35th Symposium on Mass Storage Systems and Technologies*, May 2019, pp.299–310. DOI: 10.1109/MSST.2019.00010.

[32] Wei J, Jiang H, Zhou K, Feng D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proc. the 26th IEEE Symposium on Mass Storage Systems and Technologies*, May 2010. DOI: 10.1109/MSST.2010.5496987.

[33] Guo F, Efstathopoulos P. Building a high-performance deduplication system. In *Proc. the 2011 USENIX Conference on USENIX Annual Technical Conference*, Jun. 2011.

[34] Zhang Y, Jiang H, Feng D, Xia W, Fu M, Huang F, Zhou Y. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *Proc. the 2015 IEEE Conference on Computer Communications*, Apr. 26–May 1, 2015, pp.1337–1345. DOI: 10.1109/INFOCOM.2015.7218510.

[35] Xia W, Jiang H, Feng D, Hua Y. Similarity and locality based indexing for high performance data deduplication. *IEEE Trans. Computers*, 2015, 64(4): 1162–1176. DOI: 10.1109/TC.2014.2308181.

Peng-Fei Li received his B.S. degree in computer science and technology from Huazhong University of Science and Technology, Wuhan, in 2017. He is currently a Ph.D. candidate majoring in computer system architecture at Huazhong University of Science and Technology, Wuhan. His research interests include in-memory indexes, network-attached key-value stores, and deduplication techniques.

Yu Hua received his B.S. and Ph.D. degrees in computer science from Wuhan University, Wuhan, in 2001 and 2005, respectively. He is currently a professor at Huazhong University of Science and Technology, Wuhan. His research interests include cloud storage systems, file systems, non-volatile memory architectures, etc.

Qin Cao received her B.S. degree in computer science from Central China Normal University, Wuhan, in 2017, and her Master degree in computer science and technology from Huazhong University of Science and Technology, Wuhan, in 2020. Her research interests include data deduplication techniques and persistent memory systems.