Improving the Performance of Deduplication-Based Backup Systems via Container Utilization Based Hot Fingerprint Entry Distilling

DATONG ZHANG and YUHUI DENG, Department of Computer Science, Jinan University, China
YI ZHOU, TSYS School of Computer Science, Columbus State University, USA
YIFENG ZHU, School of Electrical and Computer Engineering, University of Maine, USA
XIAO QIN, Department of Computer Science and Software Engineering, Auburn University, USA

Data deduplication techniques construct an index consisting of fingerprint entries to identify and eliminate duplicated copies of repeating data. The bottleneck of disk-based index lookup and data fragmentation caused by eliminating duplicated chunks are two challenging issues in data deduplication. Deduplication-based backup systems generally employ containers storing contiguous chunks together with their fingerprints to preserve data locality for alleviating the two issues, which is still inadequate. To address these two issues, we propose a container utilization based hot fingerprint entry distilling strategy to improve the performance of deduplication-based backup systems. We divide the index into three parts: hot fingerprint entries, fragmented fingerprint entries, and useless fingerprint entries. A container with utilization smaller than a given threshold is called a *sparse container*. Fingerprint entries that point to non-sparse containers are hot fingerprint entries. For the remaining fingerprint entries, if a fingerprint entry matches any fingerprint of forthcoming backup chunks, it is classified as a fragmented fingerprint entry. Otherwise, it is classified as a useless fingerprint entry. We observe that hot fingerprint entries account for a small part of the index, whereas the remaining fingerprint entries account for the majority of the index. This intriguing observation inspires us to develop a hot fingerprint entry distilling approach named *HID*. HID segregates useless fingerprint entries from the index to improve memory utilization and bypass disk accesses. In addition, HID separates fragmented fingerprint entries to make a deduplication-based backup system directly rewrite fragmented chunks, thereby alleviating adverse fragmentation. Moreover, HID introduces a feature to treat fragmented chunks as unique chunks. This feature compensates for the shortcoming that a Bloom filter cannot directly identify certain duplicated chunks (i.e., the fragmented chunks). To take full advantage of the preceding feature, we propose an evolved HID strategy called *EHID*. EHID incorporates a Bloom filter, to which only hot fingerprints are mapped. In doing so, EHID exhibits two salient features: (i) EHID avoids disk accesses to identify unique chunks and the fragmented chunks; (ii) EHID slashes the false positive rate of the integrated Bloom filter. These salient

This work was sponsored by the National Natural Science Foundation of China under grant 62072214, the International Cooperation Project of Guangdong Province under grant 2020A0505100040, the Science and Technology Planning Project of Guangzhou (202103000036), and the Open Project Program of Wuhan National Laboratory for Optoelectronics (2020WNLOKF006).

Authors' addresses: D. Zhang and Y. Deng (corresponding author), Department of Computer Science, Jinan University, Guangzhou, Guangdong Province, China, 510632; emails: dtz228@yeah.net, tyhdeng@email.jnu.edu.cn; Y. Zhou, TSYS School of Computer Science, Columbus State University, 4225 University Avenue, GA 31907-5645, USA; email: zhou_yi@columbusstate.edu; Y. Zhu, School of Electrical and Computer Engineering, University of Maine, Orono, ME, USA; email: zhu@eece.maine.edu; X. Qin, Department of Computer Science and Software Engineering, Auburn University, Alabama 36849, USA; email: xqin@auburn.edu.

1553-3077/2021/10-ART30 $15.00
https://doi.org/10.1145/3459626

features push EHID into the high-efficiency mode. Our experimental results show our approach reduces the average memory overhead of the index by 34.11% and 25.13% when using the Linux dataset and the FSL dataset, respectively. Furthermore, compared with the state-of-the-art method HAR, EHID boosts the average backup throughput by up to a factor of 2.25 with the Linux dataset, and EHID reduces the average disk I/O traffic by up to 66.21% when it comes to the FSL dataset. EHID also marginally improves the system's restore performance.

CCS Concepts: • **Computer systems organization → Embedded systems**; *Redundancy;*

Additional Key Words and Phrases: Data deduplication, data backup, storage system, disk bottleneck, data fragmentation

ACM Reference format:
Datong Zhang, Yuhui Deng, Yi Zhou, Yifeng Zhu, and Xiao Qin. 2021. Improving the Performance of Deduplication-Based Backup Systems via Container Utilization Based Hot Fingerprint Entry Distilling. *ACM Trans. Storage* 17, 4, Article 30 (October 2021), 23 pages.
https://doi.org/10.1145/3459626

1 INTRODUCTION

The explosive growth of data has become an increasing challenge for data storage systems [13, 27]. Data deduplication is a modern technique that can identify and store unique chunks, thereby significantly reducing the need for storage space [4, 12]. Moreover, data deduplication can minimize the network transmission of redundant data in a network storage system [23]. Due to its high efficiency, data deduplication has been widely investigated. Emerging applications of data deduplication such as I/O deduplication [18], file system deduplication [33], and memory system deduplication [15, 16, 32] trumpet the increasing popularity and importance of data deduplication. The deduplication-based backup system first divides a backup stream into fixed or variable size *chunks* [26, 29], then computes an SHA-1 digest [28] (i.e., *fingerprint*) for each chunk. And to preserve locality of backup streams, *containers* [19, 34] are used to store contiguous data chunks together with their fingerprints. When backing up a chunk, the system issues a lookup request to a *(fingerprint) index* to compare the fingerprint of the current chunk to the fingerprints of stored chunks. If a match occurs, the chunk is a duplicate copy and will be removed from the backup stream. Otherwise, the chunk is a unique one and will be stored.

Since each chunk backup will cause an index lookup to locate a matched fingerprint, when the size of the index is too large to be stored in memory, the performance of deduplication downgrades tremendously because of the poor performance of disk-based index accesses [14]. Prior studies show that accessing a disk-resident index is at least 1,000 times slower than accessing a memory-resident index [9]. Frequent disk-based index lookups are one of most important performance bottlenecks in deduplication-based backup systems [31]. For example, assuming the average size of a data chunk is 8 KB, backing up an 800-TB dataset will generate at least 2 TB of fingerprints, which will be too large to be stored in the memory [31]. There is a deduplication system employing 25 GB of memory to an index [14], and such a huge index will pose a heavy pressure on the memory and cause a large number of disk I/Os. Another bottleneck in deduplication-based backup systems is data fragmentation [17, 24]. *Fragmentation* denotes a phenomenon that logically continuous data chunks in a backup stream are physically scattered across different containers after deduplication. The heavily scattered data chunks are considered as *fragmented chunks*. Data fragmentation degrades the performance of restoring backup streams because fragmentation destroys spatial locality [21].

To avoid the preceding expensive lookup requests to the index, container metadata (i.e., fingerprints stored in containers) is prefetched from disks to memory to quickly identify contiguous

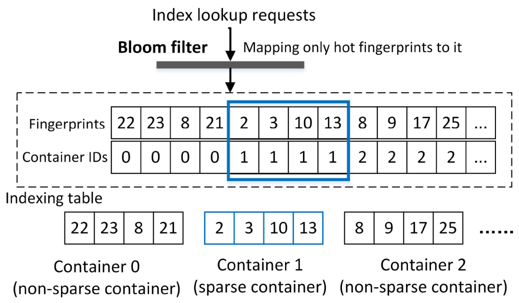

Index lookup requests
Bloom filter
Mapping only hot fingerprints to it
Fingerprints
Container IDs
Indexing table
Container 0 (non-sparse container)
Container 1 (sparse container)
Container 2 (non-sparse container)

Fig. 1. The relationship of fingerprint entries and containers.

duplicate chunks in the backup stream. For this reason, an *index* consisting of multiple *fingerprint entries* is formed to facilitate container metadata prefetching, and each *fingerprint entry* maps a fingerprint to the address of one or multiple containers [11]. Figure 1 briefly shows several key concepts (e.g., fingerprint entry, indexing table, and container) and their relationships in deduplication-based backup systems. Please note that the *indexing table* is comprised of two parts: a memory-based index and a disk-based index. In the dashed box, numbers in the first line represent fingerprints of stored data chunks, whereas numbers in the second line denote IDs of containers, each of which stores specific data chunks.

Motivation 1. Analyzing the deduplication process reveals that fingerprint entries in the indexing table can be classified into three categories: *hot fingerprint entries*, *fragmented fingerprint entries*, and *useless fingerprint entries*. Let us elaborate these terms with the example in Figure 1. At the bottom of Figure 1 are three containers, among which containers 0 and 2 are non-sparse containers, and container 1 is a sparse container. In this study, a container with utilization (detailed in Section 3) smaller than a given threshold is called a *sparse container*; fingerprint entries pointing to non-sparse containers (i.e., containers 0 and 2) are *hot fingerprint entries*, and *cold fingerprint entries* point to sparse containers, and thus we have four cold fingerprint entries 2, 3, 10, and 13 in this example. If a cold fingerprint entry's fingerprint is identical to the fingerprint of any data chunk included in future backup streams, this cold fingerprint entry is classified as a *fragmented fingerprint entry*. Otherwise, this cold fingerprint entry is a *useless fingerprint entry* since there are no matched fingerprints in chunks of future backup streams. The rationale behind classifying fingerprint entries into three categories is threefold. First, hot fingerprint entries held in memory help avoid disk-based index accessing, which speeds up duplicated chunk identifications in a backup stream during the course of deduplication. Second, keeping useless fingerprint entries in memory wastes scarce memory space since useless fingerprint entries have no contribution to duplicated chunk identification. Third, fragmented fingerprint entries reflect fragmented data chunks (please see the detailed discussion in Section 3), which should be rewritten to new containers for alleviating data fragmentation.

Motivation 2. Removing the fragmented fingerprint entries from the indexing table makes the deduplication-based backup system treat the fragmented chunks as unique chunks and directly rewrite them to new containers, thus alleviating data fragmentation. Now, we demonstrate how removing the fragmented fingerprint entries make deduplication-based backup systems directly rewrite the fragmented chunks for alleviating data fragmentation with the preceding example in Figure 1. Let us first recall some terms as follows. *Fragmentation* refers to a phenomenon that logically continuous data chunks in a backup stream are physically scattered across different containers after deduplication, and such chunks are referred to as *fragmented chunks*. A *sparse container* refers to a container with utilization smaller than a given threshold. Because container 1 in Figure 1 is a sparse container, its corresponding data chunks represented by fingerprints 2, 3, 10, and 13 will hardly appear in subsequent backup streams. For the purpose of alleviating data fragmentation, even if a few of chunks of sparse containers appear in subsequent backup streams, they will be considered as *fragmented chunks* and rewritten together with other unique chunks to new containers. For example, we assume a subsequent backup stream containing four chunks 2, A, B, and C, each of which issues an index lookup request to the indexing table (global fingerprint entries); then, only fingerprint 2 matches. Therefore, the new chunk 2 in the subsequent backup stream is a duplicated chunk since there exists a duplicate copy of the new chunk 2 in sparse container 1. Instead of deduplicating the new chunk 2, we regard the new chunk 2 as a *fragmented chunk* and rewrite it into a new container together with other unique chunks in the backup stream to alleviate data fragmentation. Note that traditional deduplication-based systems first identify the new chunk 2 as a duplicated chunk since the fragmented fingerprint entries (e.g., the entry of fingerprint 2) are not removed, then use additional rewriting algorithms to determine whether the duplicated chunk is a fragmented chunk or not, which increases computing overhead. If we remove the fragmented fingerprint entries from the indexing table, the new chunk 2 is identified as a unique chunk and directly rewritten to a new container for alleviating data fragmentation. Please note that both the stored chunk 2 in the sparse container 1 and the new chunk 2 in the backup stream are called *fragmented chunks* in this article. The fingerprint entries (e.g., the entry of fingerprint 2) of fragmented chunks are referred to as *fragmented fingerprint entries*.

Inspired by the preceding motivations, in this study we first propose a container utilization (detailed in Section 3) based hot fingerprint entry distilling approach—*HID*—to alleviate the disk I/O bottleneck and reduce data fragmentation. HID leverages container utilization to identify hot and cold fingerprint entries. HID separates cold fingerprint entries from the indexing table at the end of a backup process. Then in the next backup process, only hot fingerprint entries will be used to construct the indexing table (containing a memory-based index and a disk-based index). In doing so, HID improves the hit ratio of memory-based index lookups since all fingerprint entries in the indexing table are hot fingerprint entries, thereby avoiding frequent disk-based index access. Moreover, HID reduces data fragmentation because the separation of fragmented fingerprint entries enables a deduplication-based backup system to treat fragmented chunks as unique chunks to directly rewrite them into containers.

Motivation 3. It is noteworthy that we propose a salient featurecalled *FTU* (Fragmented chunks are Treated as Unique chunks). The mechanism of FTU is that HID only keeps hot fingerprint entries in the indexing table, leaving no fragmented fingerprint entries in the index table. When performing an index lookup on a fragmented chunk, there is no matching entry in the indexing table, so fragmented chunks are regarded as unique chunks. FTU is introduced to offset the deficiency of a Bloom filter. The Bloom filter is a deduplication optimization technique used to identify unique chunks in backup streams. However, the Bloom filter fails to identify duplicate chunks in backup streams due to the false-positive issue (i.e., the hash collision) [5, 22, 34]. Without FTU, fragmented chunks may degrade the performance of a deduplication-based backup system. More specifically, fragmented chunks are a subset of duplicate chunks. Each fragmented chunk in backup streams is not identified by the Bloom filter and will issue an index lookup request to the indexing table (see Figure 1), which may cause disk-based index accesses. Fortunately, the Bloom filter mated with FTU (by map only hot fingerprints to this Bloom filter) regards fragmented chunks as unique chunks and can effectively identify these fragmented chunks, thereby avoiding the indexing table lookups.

To exploit the full advantages of FTU and the Bloom filter, we finally propose an evolved container utilization based hot fingerprint entry distilling strategy—*EHID*—to improve our initial HID. EHID embeds a Bloom filter and maps only hot fingerprints into the Bloom filter. In doing so, EHID exhibits three merits. First, EHID effectively avoids disk-based index accesses when identifying both unique and fragmented chunks. Second, the false-positive rate of the Bloom filter is reduced by mapping only hot fingerprints into the integrated Bloom filter (please see Section 5.3). Third, EHID achieves a considerable performance improvement in deduplication-based backup systems. To sum up, this article makes the following contributions:

- We obtain a handful of compelling and valuable findings. First, to reduce disk-based index lookups and alleviate data fragmentation, we show that global fingerprint entries can be divided into three categories: hot fingerprint entries, useless fingerprint entries, and fragmented fingerprint entries. Useless fingerprint entries are useless to identify duplicate chunks in the backup stream, so caching them in memory is unwise. Separating useless fingerprint entries alleviates the problem of disk-based index lookups. The separation of fragmented fingerprint entries enable a deduplication-based backup system to regard fragmented chunks as unique chunks to directly rewrite them to new containers, thus alleviating data fragmentation. Second, we reveal that container utilization is an effective indicator for facilitating the three-category classification of fingerprint entries. Fingerprint entries pointing to non-sparse containers are hot fingerprint entries, and fingerprint entries pointing to sparse containers are useless or fragmented fingerprint entries.
- We propose *HID* to deal with the disk-based index lookup bottleneck and alleviate data fragmentation in deduplication-based backup systems.
- We propose an evolved HID called *EHID*. EHID embeds a Bloom filter and maps only hot fingerprints to the embedded Bloom filter. In doing so, EHID significantly avoids expensive disk-based index accesses. Extensive experimental results demonstrate that EHID achieves considerably higher performance.

The rest of this article is organized as follows. Section 2 summarizes the related work. Section 3 introduces related background knowledge. Section 4 presents our motivation and analyses of the fingerprint entry category conversion. Section 5 demonstrates our design and implementation. Comprehensive experiments are performed to evaluate the proposed approaches in Section 6. Section 7 concludes the article.

2 RELATED WORK

Many studies have been conducted to handle the bottleneck of disk-based index lookups. For example, a Bloom filter [5] is used by **exact deduplication (ED)** approaches [22, 34] to identify unique chunks. However, the efficiency of the Bloom filter declines as the amount of backup data increases. This is because the false-positive rate [5, 22, 34] of the Bloom filter becomes larger when its space usage grows. Near-ED methods perform deduplication by coarsely sampling a portion of all fingerprint entries (e.g., random sampling, uniform sampling) [3, 14, 20, 31]. Since the number of sampled fingerprint entries is much smaller than the total number of fingerprint entries, near-ED methods reduce the number of disk accesses caused by index lookups. Although the ED and near-ED approaches reduce the number of disk I/Os, both of them involve a large number of useless fingerprint entries and fragmented fingerprint entries, which can cause frequent disk accesses during index lookups. To alleviate data fragmentation, HAR (i.e., a History-Aware Rewriting algorithm) makes use of container utilization to identify and rewrite fragmented chunks [10]. Please note that rewriting fragmented chunks refers to allowing a small number of duplicated chunks to be stored on disk. Unfortunately, HAR ignores the fact that container utilization can facilitate categorizing fingerprint entries. In this research, we leverage container utilization to classify fingerprint entries, aiming to speed up index lookups and alleviate data fragmentation. To make the related work more clear and comprehensive, the following two paragraphs introduce each work in more detail.

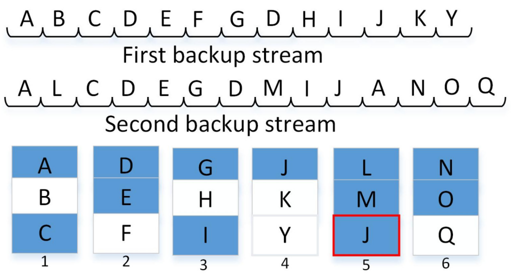


Fig. 2. Example of deduplication for two backup streams.

To reduce accesses to the disk-based index, many deduplication methods have been proposed. Zhu et al. [34] use a Bloom filter to identify unique chunks and employ a container to preserve the locality of the backup streams, thereby overcoming the disk bottleneck. MFDedup [35] maintains the locality of backup workloads by using a data classification approach to generate an optimal data layout. However, the false-positive rate of the Bloom filter [5] increases sharply as the spatial usage of this Bloom filter grows. Sparse indexing approaches [20] randomly sample a percentage of fingerprint entries from all fingerprint entries of stored chunks (intensive index) to construct the sparse index. The sparse indexing methods reduce the memory overhead of the index and improve the backup throughput. However, its deduplication ratio relies on the locality of the backup streams and sampling ratio. Guo and Efstathopoulos [14] apply the sampling method from logical locality to physical locality. Extreme Binning [3] and ChunkStash [8] take advantage of similarity theory of files instead of the locality of the backup streams. However, it is sensitive to the similarity of files. Xia et al. [31] take advantage of both locality and similarity, and it works well for workloads with weaker or lower similarity. However, the indexing table constructed by the preceding contains a large number of cold fingerprint entries, and the cold fingerprint entries result in frequently accessing a disk-based index when performing index lookups.

Since fragmentation greatly degrades the restore performance, many rewriting algorithms for mitigating fragmentation have been proposed. Most rewriting algorithms [6, 17, 19, 25] allocate small-size memory to buffer the consecutive data chunks in a backup stream. The sequence of data chunks in the buffer is called a *logical sequence*. In contrast, a sequence of data chunks located on disk is called a *physical sequence*. If the logical sequence is significantly different from the physical sequence, the rewriting algorithms determine that the buffer has many fragmented chunks and rewrites them. However, since the allocating memory only buffers a fraction of a backup stream, the fragmented chunks are identified inaccurately. To deal with this problem, HAR [10] uses the global information of previous backup streams to predict the fragmentation of latter backup streams. HAR overcomes the shortcoming of the other rewriting algorithms and improves both the restore performance and the deduplication ratio. Unfortunately, HAR fails to consider that the significant indicator—container utilization—reflects the fingerprint entries' categories. In addition, the overhead of updating fragmented fingerprint entries in HAR diminishes the backup performance.

3 BACKGROUND

Fragmentation. Figure 2 shows that by dealing with the first backup stream, the unique chunks are aggregated to containers in sequence. The data chunks of the first backup stream are stored in containers 1, 2, 3, and 4. When the second backup stream arrives, chunk A is identified as duplicated. Thus, A is not stored, and it is replaced by a pointer to the shared chunk A located in container 1. The next chunk L is identified as a unique chunk, and thus it is stored in container 5 in this example.

Following this process, the remaining data chunks of the backup stream are deduplicated one by one. At the end, all contiguous data chunks of the second backup stream are scattered in different containers, which is called *data fragmentation*.

Container utilization. Figure 2 demonstrates that for the first backup stream, container 1 has three shared chunks: A, B, and C. However, for the second backup stream, container 1 only has two shared chunks: A and C. There is a high probability that container 1 only has less than two shared chunks for all latter backup streams, since the backup data is continuously modified (deleting/adding/modifying the data chunks). In other words, the number of shared chunks in a container is gradually decreased with the growth of the backup version in the backup and archive systems. Such effect is reflected by the container utilization and can be expressed as the following Equation (1). *Container utilization* is defined as

$$CU_i^k = R_i^k / S, \tag{1}$$

where k represents the kth backup stream, i represents the ith container, S denotes the container size (the number of data chunks), R_i^k denotes the number of shared chunks in the ith container for the kth backup stream, and CU_i^k denotes the container utilization of the ith container for the kth backup stream.

As shown in Figure 2, since $CU_1^1 = 3/3 = 100\%, CU_1^2 = 2/3 = 66.66\%$, we can conclude that the higher the container utilization, the more frequent this container is accessed during data backup and restore. Container utilization has historical inheritance characteristics [10]. For a container i, its container utilization decreases as backup versions (streams) increase. In other words, $CU_i^1 \geq CU_i^2 \geq \cdots \geq CU_i^n$. Therefore, for a typical backup scenario, the container utilization does not increase unless a rare rollback occurs [10]. However, in those rare cases, the growth of container utilization has negligible impact on the overall system performance.

Additionally, the intra-backup duplication has no impact on the container utilization. For example, as we can see from Figure 2, container 2 contains three data chunks: D, E, and F. Although the first backup stream contains data chunks E, F and two duplicate copies of D, R_2^1 is still calculated as 3. This indicates that no matter how many times E, F, or D repeatedly appears in the first backup stream, R_2^1 remains unchanged. Because S is a fixed value, according to Equation (1), the container utilization CU_2^1 also remains unchanged.

Sparse container. The sparse container represents the container whose utilization is below a given threshold. For example, if the threshold P is set to 70%, for the second backup stream, the container 1 is a sparse container since $CU_1^2 = 66.66\% < P$. According to its historical inheritance characteristic, if a specific container is identified as a sparse container in the previous backup stream, the container will still be a sparse container in subsequent backup streams. This is a basic precondition of our approaches.

Rewriting fragmented chunks. To improve the restore performance, HAR [10] rewrites the fragmented chunks and reduces the number of sparse containers. This is shown in Figure 2. For example, assuming the sparse threshold $P = 50\%$, since $CU_4^2 = 1/3 = 33.33\% < P$, container 4 is a sparse container. When processing the second backup stream, the duplicated chunk J (chunk J in the second backup stream) is identified as a fragmented chunk, since the fingerprint entries of the shared chunk J points to the sparse container 4. Therefore, the fragmented chunk J is written to the new container 5 together with unique chunks L and M. When restoring the second backup stream, chunk J is directly obtained from container 5 cached in the memory, because container 5 has been read from disk to memory when restoring the former chunk L. In contrast, if the fragmented chunk J is not rewritten, reading container 4 is inevitable to obtain chunk J, which incurs a disk I/O and damages the restore performance.

Detailed data deduplication process. As shown in Figure 3, the indexing table (containing a memory-based index and a disk-based index) consists of all fingerprint entries of the stored chunks. Each fingerprint entry maps a fingerprint to the corresponding container ID (logical address). The containers are located in disk storage, and each container preserves corresponding fingerprints. The recipes is used to record the logical chunk sequence for restoring data. The active container is used to store the unique chunks and rewritten chunks. The byte stream (backup stream) has already been divided into chunks, and the fingerprints have been computed. When backing up a new chunk, the system first goes to the container metadata cache to locate an identical fingerprint. If the fingerprint does not hit in the cache, then the system will generate a lookup request for the indexing table, which may issue accessing the disk-based index. At the end of searching the index, the attribute of new chunks is determined. If the new chunk is a unique chunk, the system writes it into the active container, and generates a new fingerprint entry and inserts it to the indexing table. If the new chunk is a duplicated chunk, it will not be stored. If the new chunk is a fragmented chunk, the system rewrites it to the active container and then updates the fragmented fingerprint entry (making the entry point to the active container instead of the sparse container). Note that updating a fragmented fingerprint entry may require accessing the disk-based index, because it has to look up the indexing table again to locate the fragmented fingerprint entry and then update the entry's value (container ID).

Restore process. After the deduplication process, the data chunks of a file may be completely disordered and scattered in various containers across all disks, and the fingerprint entries in the indexing table are also disordered. Therefore, the indexing table cannot be leveraged to restore the data due to lacking the sequence information of data chunks. File recipes stored on disk are employed to restore data instead of using the indexing table. The file recipe consists of fingerprints and the physical addresses of the corresponding data chunks. The sequence of fingerprints in the file recipe is consistent with the sequence of data chunks in the backup stream. Each backup version generates a specific file recipe. If the *kth* backup version is required to restore, the system will first analyze the metadata of the file recipes, obtain the stored path of the kth file recipe, load the file recipe into memory, and read the data chunks from disk to restore the data by using the file recipe information. Because the file recipes are independent of the indexing table, deleting any of the fingerprint entries from the indexing table does not have any impact on the restore process.

4 MOTIVATION DISCUSSION AND FINGERPRINT ENTRY CATEGORY CONVERSION ANALYSIS

4.1 Motivation Discussion

Looking up the indexing table and the increasing fragmented chunks cause frequent accesses to the disk-based index and the data fragmentation, respectively, leading to a major performance bottleneck in deduplication-based backup systems. Our key idea is that separating useless fingerprint entries has no impact on identifying and eliminating duplicated chunks, and separating fragmented fingerprint entries makes the systems directly rewrite fragmented chunks and reduces data fragmentation. Our key observation is that a sparse container reflects useless and fragmented fingerprint entries. For example, Figure 3 shows that fingerprint entries whose fingerprints are 2, 3, 10, and 13 in the indexing table are cold fingerprint entries, because they all point to container 1, which is assumed to be a sparse container.

These facts enable us to classify the fingerprint entries as hot, fragmented, and useless, and fragmented and useless fingerprint entries constitute cold fingerprint entries. For example, in Figure 3, the fingerprint entries in the indexing table pointing to sparse containers are classified as cold fingerprint entries, and the remaining are classified as hot fingerprint entries. More specifically,

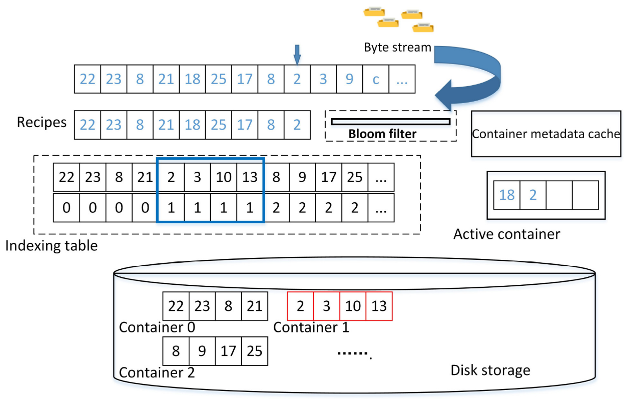


Fig. 3. The detailed process of deduplication.

fingerprint entries whose fingerprints are 10 and 13 are useless fingerprint entries, because they are not accessed in the subsequent backup process due to the characteristics of data backup scenarios (detailed in Section 3.2). Fingerprint entries 2 and 3 are fragmented fingerprint entries, because there are duplicated fingerprints (chunks) 2 and 3 in the subsequent byte stream but their shared chunks are located in the sparse container 1. In this article, we call chunks like the duplicated chunks 2 and 3, as well as the shared chunks 2 and 3, *fragmented chunks*.

Separating useless fingerprint entries can effectively save memory space, and thus it is always performed. However, separating fragmented fingerprint entries is optional. In the design of HID, we choose to separate fragmented fingerprint entries for the following three reasons:

- Separating fragmented fingerprint entries introduces an FTU feature. The FTU feature denotes that separating fragmented fingerprint entries make deduplication-based backup systems treat the fragmented chunks as unique chunks. For example, in Figure 3, after entries of fingerprints 2 and 3 are separated, fragmented chunks 2 and 3 in the byte stream are identified as unique chunks by traversing the indexing table.
- Separating fragmented fingerprint entries makes deduplication-based backup systems directly rewrite fragmented chunks, and such a direct rewriting mechanism alleviates the data fragmentation and avoids the overhead of updating fragmented fingerprint entries. For instance, thanks to the FTU feature, the fragmented chunks 2 and 3 in the byte stream (see Figure 2) are transformed into unique chunks (FTU's unique chunks). The deduplication-based backup system directly writes these FTU's unique chunks to the active container and generates new fingerprint entries for these chunks rather than updating the fragmented fingerprint entries that are separated.
- The operation of dividing the fingerprint entries pointing to sparse containers (i.e., cold fingerprint entries) into useless fingerprint entries and fragmented fingerprint entries is expensive. To avoid this overhead, the fingerprint entries pointing to sparse containers are directly separated from all fingerprint entries.

Integrating a Bloom filter and FTU. FTU is complementary to a Bloom filter. A Bloom filter in a deduplication-based backup system can efficiently identify unique chunks by mapping all fingerprints into itself. However, it lacks the ability to identify duplicated chunks due to its false positive [5, 22]. Specifically, duplicate chunks in a byte stream must look up the index, which may result in disk-based index accesses. Fortunately, with FTU integrated, the Bloom filter is enabled to identify some duplicated chunks, because FTU treats all fragmented chunks as unique chunks. Note that the fragmented chunks belong to duplicated chunks, and fragmented chunks are only a small part of duplicated chunks. In summary, the combination of the Bloom filter and FTU further prevents a deduplication-based backup system from the disk-based index accesses. At the level of code implementation, As long as we map the hot fingerprints instead of all fingerprints to the integrated Bloom filter, the Bloom filter and FTU can be integrated very well.

Based on the preceding analysis, we propose EHID to further improve the backup performance of deduplication-based backup systems by integrating a Bloom filter and FTU. EHID places the Bloom filter between the container metadata cache and the indexing table, and it maps only hot fingerprints into this Bloom filter (see Figure 3). In doing so, most of all unique chunks (normal and FTU's unique chunks) are identified by the Bloom filter instead of the indexing table. For example, since cold fingerprint entries (i.e., the entries of fingerprints 2, 3, 10, and 13) are not mapped to the Bloom filter, when processing new chunks 2 and 3 in the byte stream, the Bloom filter identifies these chunks as unique ones. Thereby, this avoids looking up the indexing table to determine the attributes (unique, duplicated, or fragmented) of these two chunks. Note that the traditional methods will continue to search the indexing table, because they map the cold fingerprints (i.e., 2, 3, 10, and 13) to the Bloom filter considering that the Bloom filter cannot treat new chunks 2 and 3 as the unique chunks. In addition to the mutual gain of FTU and the Bloom filter, mapping only hot fingerprints instead of all fingerprints into the Bloom filter decreases the false-positive rate.

4.2 Fingerprint Entry Category Conversion Analysis

There are still two key issues that HID and EHID need to solve. Can a useless or fragmented fingerprint entry can be reclassified as a hot one as the workload evolves? How do HID and EHID deal with this situation when it happens? To answer these two questions, we conduct the following analysis. Note that the indexing table (global fingerprint entries) is created to quickly identify duplicated chunks in the backup streams. Consider the following typical data backup scenarios.

An engineer creates a file, which is represented by $\{A1, A2, A3, A4\}_0$, the subscript 0 indicates that the file is the initial version, and each character in "{}" represents a data chunk, the $\{A1, A2, A3, A4\}_0$ means that the file contains four data chunks: $A1, A2, A3$, and $A4$. To protect the file, the engineer uses a deduplication-based backup system to make the first backup. Since there are no chunks stored in the backup storage at this moment, the indexing table is empty. The indexing table is represented by $(\varnothing)_0$, in which the subscript 0 indicates that the indexing table has not been updated yet, and each character in "()" represents the fingerprint of an entry. When data chunks $A1$ through $A4$ are backed up, the system searches the indexing table $(\varnothing)_0$. As the indexing table is empty, these data chunks are identified as unique chunks and stored to disks. Then, the system creates the entries of fingerprints a1 through $a4$ for the four chunks stored. At this time point, the indexing table is updated from $(\varnothing)_0$ to $(a1 - a4)_1$.

Later on, the engineer modifies the file and the file becomes $\{A1, A2, B1, A4\}_1$. At this time, he makes the second backup. The system traverses $(a1 - a4)_1$ and finds the fingerprints $a1, a2$, and $a4$, respectively. Therefore, data chunks $A1, A2$, and $A4$ are identified as duplicated chunks. The data chunk $B1$ is identified as a unique chunk, because there is no corresponding fingerprint in $(a1 - a4)_1$. Then the system creates a fingerprint entry $b1$ for data chunk $B1$ and inserts $b1$ to the indexing table. Now, the indexing table is updated from $(a1-a4)_1$ to $(a1-a4, b1)_2$. We can find that during the second backup, only the fingerprint entries $a1, a2$, and $a4$ in $(a1 - a4)_1$ are helpful for identifying the duplicated chunks $A1, A2$, and $A4$ in $\{A1, A2, B1, A4\}_1$. In other words, removing the entry of fingerprint $a3$ from indexing table $(a1 - a4)_1$ has no impact on the deduplication process.

After that, the engineer modifies the file again and the file becomes $\{A1, A2, B1, C1, C2\}_2$. At this time, she makes the third backup. In the same way, only the entries of fingerprints $a1, a2$, and $b1$ in $(a1 - a4, b1)_2$ are helpful for identifying the duplicated chunks $A1, A2$, and $B1$ in $\{A1, A2, B1, C1, C2\}_2$. Therefore, it creates no side effects to remove fingerprint entries $a3$ and $a4$ from $(a1 - a4, b1)_2$.

In the preceding backup example, the indexing table is updated in the following sequence:

$$\begin{aligned} &(\varnothing)_0 \rightarrow (a1, \underline{a2}, a3, a4)_1 \rightarrow (a1, \underline{a2}, \underline{a3}, a4, b1)_2 \\ &\rightarrow (\underline{a1}, \underline{a2}, \underline{a3}, a4, b1, c1, c2)_3 \rightarrow \cdots \rightarrow (\underline{a1}, \underline{a2}, \underline{a3}, \\ &\underline{a4}, \underline{b1}, c1, c2, \ldots, xm)_n, \end{aligned} \tag{2}$$

where n is the number of backups and underlined letters indicate useless fingerprint entries. This example shows that although the total number of fingerprint entries increases with backup versions, the number of useless fingerprint entries also increases accordingly. Previous studies [6, 7, 11] show that as more backups are made, the number of fragmented chunks increases and the number of fragmented fingerprint entries also increases. Therefore, we can infer that the number of hot fingerprint entries remains at a stable level or grows slowly as the backup versions increase.

In a typical data backup scenario, with the increase of backup versions, the changing trend of the three fingerprint entries are as follows:

- *Hot fingerprint entry:* If a fingerprint entry is classified as hot in the current backup, it is possibly a hot fingerprint entry or becomes a useless or a fragmented fingerprint entry in subsequent backups.
- *Useless fingerprint entry:* If a fingerprint entry is classified as useless in the current backup, it is still a useless fingerprint entry in subsequent backups.
- *Fragmented fingerprint entry:* If a fingerprint entry is classified as fragmented in the current backup, it is still a fragmented fingerprint entry in subsequent backups.

Due to the characteristics of a typical data backup scenario, a cold fingerprint entry (i.e., a useless fingerprint entry and a fragmented fingerprint entry) is not reclassified as a hot fingerprint entry anymore. Therefore, both HID and EHID work efficiently in any backup versions.

5 DESIGN AND IMPLEMENTATION

5.1 Architecture Overview

Figure 4 depicts the overall architecture and the workflow of our EHID strategy. The entire system can be divided into three large modules: a **Fingerprint Lookup Module (FLM)**, a **Container Utilization Monitoring Module (CUMM)**, and an **Index Separating Module (ISM)**. These modules work collaboratively by repeating the following three steps:

- At step 1, FLM is responsible for identifying unique chunks and duplicated chunks when each new data chunk is backed up. Meanwhile, CUMM computes the utilization of each container by recording which data chunk belongs to which container in the backup process and then obtains a list of sparse containers. The list is used for distinguishing hot and cold fingerprint entries.
- At step 2, ISM identifies cold fingerprint entries and hot fingerprint entries from all fingerprint entries by checking whether a fingerprint entry points to a sparse container that is recorded on the list. If the fingerprint entry points to a sparse container, it is a cold fingerprint entry. Otherwise, it is hot. In this way, ISM separates cold fingerprint entries and hot fingerprint entries.

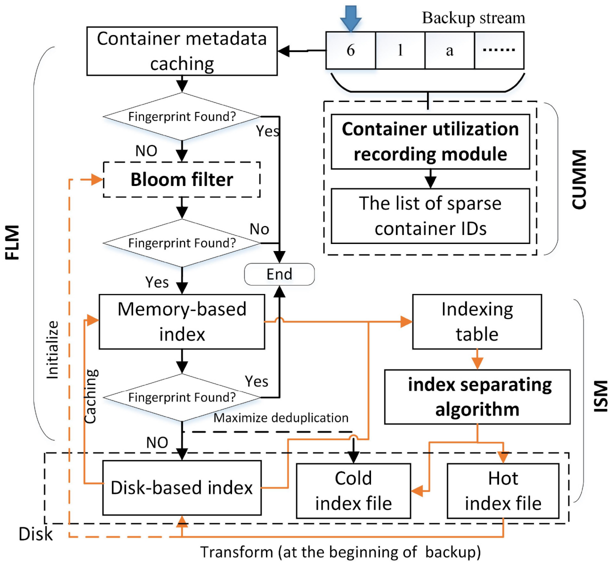


Fig. 4. Architecture overview of the EHID-based deduplication-based backup system.

- At step 3, hot fingerprint entries (the hot index file) become the indexing table consisting of the memory-based and disk-based index. Hot fingerprints are used to initialize the Bloom filter at the beginning of the next backup. Finally, the system goes back step 1 and repeats the preceding process.

The key process. As the orange arrows shown in Figure 4, the memory-based and disk-based index constitute the indexing table. As more data are backed up, the number of all fingerprint entries in the indexing table becomes greater too, even though a part of the fingerprint entries gradually turn into cold fingerprint entries. The index separating algorithm (see Figure 4) divides the fingerprint entries in the indexing table into a cold index file and a hot index file at the end of every backup process. In this way, an increasing number of cold fingerprint entries is saved to the cold index file, and the number of hot fingerprint entries in the hot index file is kept small. At the beginning of every backup, only the hot index file is used to construct the indexing table (the disk-based index and the memory-based index), and this file is also used to initialize the Bloom filter. The Bloom filter is re-initialized at the beginning of each backup. FLM does not query the cold index file unless maximizing the deduplication ratio is required in our design (note that traditional deduplication-based backup systems are provided with only the FLM module).

The cold index file and hot index file are stored on disks. At the initial stage of a specific backup, the system takes the hot index file as the disk-based index and reads a portion of the disk-based index into memory. At this time, the indexing table equals the disk-based index, and both the indexing table and disk-based index contain the fingerprint entries in the memory-based index. During the backup process, new fingerprint entries are continuously generated and inserted into the memory-based index. If the allocated memory size is fixed, the inserted fingerprint entries will trigger replacement of the fingerprint entry in memory. At this stage, the disk-based index may not include all of the memory-based index. Therefore, the indexing table used for deduplication should contain both the disk-based index and memory-based index as Figure 4 shows. After the completion of the current backup, ISM separates the indexing table as a cold index file and a hot index file and stores them on disk again. Therefore, the hot index file and the cold index file are updated for every backup version. Additionally, during the backup process, the indexing table is updated continuously for the subsequent backup stream.

The overhead of CUMM and ISM is low. CUMM uses a 64-bit integer array to record sparse containers. One backup has about 3,500 containers in our experiments. Assuming that all containers are sparse containers, CUMM only takes 27.34 KB of memory space and a little computing overhead to operate. ISM separates cold and hot fingerprint entries by going through the indexing table just once. Suppose that the number of all fingerprint entries is N and the time complexity of this separation process is $O(N)$. It is noteworthy that the time complexity is also $O(N)$ for a lookup request for accessing a disk-based index. Moreover, separating the index can be done offline to minimize the overhead before starting the next backup.

5.2 The Algorithm of Distilling Hot Fingerprint Entries

Algorithm 1 demonstrates the basic working process of HID. It is outlined as the following three important steps:

- When the nth backup starts, hot fingerprint entries distilled by the separates function in the $(n-1)th$ backup (e.g., $hIndex_{n-1}$) construct the indexing table in the current backup (e.g., $glIndex_n$), regardless of $coIndex_{n-1}$.
- In the backup process, the lookup function inserts the fingerprint entries of unique chunks (e.g., $entry_{uniq\_chunk}$) into the indexing table (e.g., $glIndex_n$). Meanwhile, a small part of all fingerprint entries in the indexing table is converted to cold fingerprint entries.
- When the backup process completes, the separates function distills hot fingerprint entries (e.g., $hIndeX_n$) and stores them to disk for the next backup. When the next backup begins, Algorithm 1 goes back to the first step once again.

5.3 EHID

Figure 4 outlines the architecture and workflow of EHID. EHID has two advantages:

- EHID takes full use of FTU and the Bloom filter to effectively identify the normal unique chunks and FTU's unique chunks (i.e., fragmented chunks). This design prevents deduplication-based backup systems from accessing a disk-based index when identifying unique chunks in the backup stream.
- The Bloom filter employed by EHID works in high efficiency. The false-positive rate of the Bloom filter increases with the growth of its space usage. Fortunately, EHID separates the cold fingerprint entries in a timely manner while minimizing the number of hot fingerprint entries. Such effects reduce the space usage taken by a hot fingerprint of the Bloom filter.

5.3.1 Analyses of Accessing the Disk-Based Index. The following formally analyzes the efficiency of EHID. Suppose that the number of data chunks in one backup stream is fixed, and each unique data chunk must trigger a request to look up the disk-based index. On the opposite side, each duplicated chunk may trigger a search of the disk-based index with similar probability, because an index lookup request of a duplicated chunk can be served in the container metadata cache or memory-based index. The average probability of each duplicated chunk triggering a disk-based index lookup request is supposed to be equal and denoted by P_{find}. Suppose that $uniqs$ and $dups$ denote the number of unique chunks and duplicated chunks, respectively. Then, for the traditional deduplication-based backup systems, the number of lookup requests for accessing the disk-based index in one backup stream is defined as

$$disk\_lookups_{tradis} = uniqs + dups * p_{find}. \tag{3}$$

ALGORITHM 1: Hot Fingerprint Entries Distilling Algorithm

```
Input: $list_n$; //list of sparse container IDs in the $nth$ backup
       $glIndex_n$; // all fingerprint entries in the $nth$ backup stream
Output: $hIndex_n$; // hot fingerprint entries in the $nth$ backup stream
        $coIndex_n$; // cold fingerprint entries in the $nth$ backup stream
1:  function INIT()
2:      $glIndex_n \leftarrow hIndex_{n-1}$;
3:  end function
4:  function LOOKUP( )
5:      while ($new\_chunk \neq end_{backupStream}$) do
6:          if ($new\_chunk = uniq\_chunk$) then
7:              $insert(glIndex_n, entry_{uniq\_chunk})$;
8:          end if
9:          $new\_chunk \leftarrow new\_chunk- > next$;
10:     end while
11: end function
12: function SEPARATES()
13:     get $first$ in $glIndex_n$;
14:     while ($first \neq end_{glIndex_n}$) do
15:         if $first- > data.value$ in $set(list_n)$ then
16:             $write(first- > data, coIndex_n)$;
17:         else
18:             $write(first- > data, hIndex_n)$;
19:         end if
20:         $first \leftarrow first- > next$;
21:     end while
22: end function
```

We assume that the false-positive rate of the Bloom filter is 0; for traditional deduplication-based backup systems configured with a Bloom filter, the number of disk-based index lookup requests is defined as

$$disk\_lookups_{tradis+Bloomfilter} = dups * p_{find}. \tag{4}$$

For EHID, the number of disk-based index lookup requests is defined as

$$disk\_lookups_{EHID} = dups^{'} * p_{find}. \tag{5}$$

Note that $dups^{'} < dups$, because EHID treats some duplicated chunks (i.e., fragmented chunks) as unique chunks (i.e., the FTU feature). In other words, the number of unique chunks identified by EHID is larger than that identified by traditional deduplication-based backup systems (i.e., $uniqs' \geq uniqs$). Correspondingly, the number of duplicated chunks identified by EHID is smaller than that of traditional deduplication-based backup systems (i.e., $dups' \leq dups$). Therefore, we can derive that $disk\_lookups_{tradis} \geq disk\_lookups_{tradis+Bloomfilter} \geq disk\_lookups_{EHID}$, and this expression demonstrates that EHID requires the minimal number of lookup requests for accessing a disk-based index.

We will give an example to illustrate the reason the EHID strategy can leverage a Bloom filter to identify fragmented chunks and avoid index access on disks, thus improving the backup throughput. Please note that the Bloom filter employed by traditional approaches cannot identify fragmented chunks. Suppose that there is a fragmented chunk J1 stored on disk, and a new backup

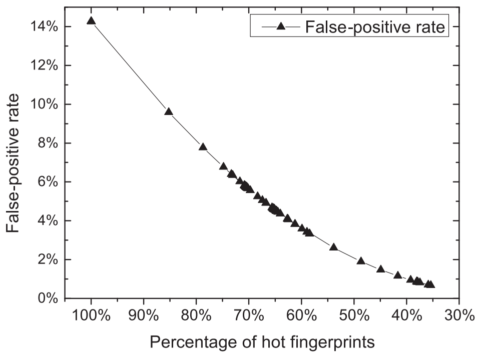


Fig. 5. False-positive rate vs. percentage of hot fingerprint entries.

chunk J2 is a duplicated chunk of J1. To alleviate the data fragmentation, J2 has to be written to disk although it is a duplicated chunk. Since EHID does not map the fragmented fingerprints to the Bloom filter (to integrate the Bloom filter and FTU), the system cannot find the mapping bit of the chunk J1's fingerprint. Therefore, the new chunk J2 is identified as a unique chunk and will be written to disk. If the fragmented fingerprints are mapped to the Bloom filter, the mapping bit of chunk J1 will be located. Due to the false positive of the Bloom filter, the system has to further check the memory-based index and disk-based index in sequence to identify the attribute of chunk J2 including the unique chunk, duplicated chunk, and fragmented chunk. Looking up the fingerprint (entry) may trigger disk I/Os in the following steps.

5.3.2 False-Positive Rate of EHID's Bloom Filter. Another fascinating feature of EHID is that it reduces the false-positive rate of the integrated Bloom filter. The calculation formula of the false-positive rate (*FPR*) of the Bloom filter can be expressed as

$$FPR = (1 - e^{(-kn/m)})^k, \tag{6}$$

where k denotes the number of hash functions, m is the size of the Bloom filter (bits), and n is the number of mapping fingerprints. Assuming a 128-KB Bloom filter with k hash functions and 250,000 fingerprints, the false-positive rate is 14.28%, implying that there are about 14 lookup requests for accessing the indexing table to process every 100 chunks. EHID only maps hot fingerprints (37.8% of all fingerprints on the Linux trace) to the Bloom filter, and accordingly the false-positive rate is decreased to 0.78%. This example shows that accesses to the indexing table are significantly reduced. Figure 5 shows the relationship between the false-positive rate and the percentage of hot fingerprint entries. It is demonstrated that when the percentage is decreased from 100% to 37.8%, the false-positive rate of the Bloom filter is reduced from 14.2% to 0.8%. Please note that the percentage of hot fingerprint entries on the x-axis is generated by different backup versions from 1 to 213 of the Linux dataset.

6 EXPERIMENT EVALUATION

6.1 Evaluation Environment

In this section, we conduct a group of experiments to evaluate the performance of the distilling approaches of our hot fingerprint entries. We implemented both HID and EHID on an open source project named *destor* [11]. To quantitatively evaluate the performance, we leverage four metrics, including the memory overhead of the indexing table, backup performance, restore performance, and storage overhead (more intuitive than the deduplication ratio). The ED method identifies and

Table 1. Characteristics of Workloads Used for Evaluation

Dataset Name	TS^1	DR^2	ACS^3	Versions (#)
Linux	99 GB	97.85%	3.1 KB	213
FSL	1.95 TB	98.73%	4.4 KB	173

[1] TS stands for the total size of the trace.
[2] DR stands for the deduplication ratio of the trace.
[3] ACS stands for the average chunk size of the trace.

deletes every duplicated chunk and does not use any rewriting algorithms. It serves as the baseline for performance evaluation. Furthermore, the performance of HID and EHID are compared against the state-of-the-art algorithm HAR [10]. Since HID and EHID only differ in the backup performance, the other three metrics (the memory overhead of the indexing table, restore performance, and storage overhead) of the experimental results of HID are omitted.

The experimental operating system is CentOS release 7.4 (Linux version 3.10.0-693.11.1. el7.x86-64). The container size is 4 MB (containing about 1,000 data chunks). The sparse threshold is a very important parameter for both HAR and EHID.

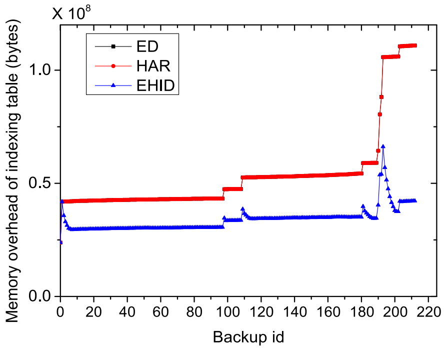
The number of sparse containers grows with the increase of the sparse threshold. This results in more fragmented chunks, useless chunks, and cold fingerprint entries. Because both EHID and HAR improve the restore performance by rewriting fragmented chunks, a higher sparse threshold results in that both EHID and HAR rewrite more fragmented chunks, and thus both of them achieve a higher restore performance and a lower deduplication ratio. Additionally, in contrast to HAR, EHID separates cold fingerprint entries from the indexing table. Therefore, a higher sparse threshold results in that EHID separates more cold fingerprint entries, which will further reduce the memory overhead of the indexing table. As a result, EHID further improves the backup performance, whereas HAR cannot improve. When the threshold is decreased to a lower value, it will generate an opposite process. It has been proved by HAR that setting the threshold at 50% can strike a good balance between the deduplication ratio and restore performance. Therefore, to have a fair comparision with HAR, we set the threshold at 50% across all of the experiments in this article.

The Linux dataset and the FSL dataset [1, 2] are commonly used public datasets [30]. The characteristics of these two datasets are shown in Table 1. Since the FSL trace does not have the byte stream of the data chunks, we cannot evaluate the actual throughput on the FSL trace like on the Linux trace. We analyze the disk I/O behaviors caused by index lookup in the deduplication process and use the number of disk I/Os to measure the throughput and evaluate the efficiency of our approaches on the FSL dataset.

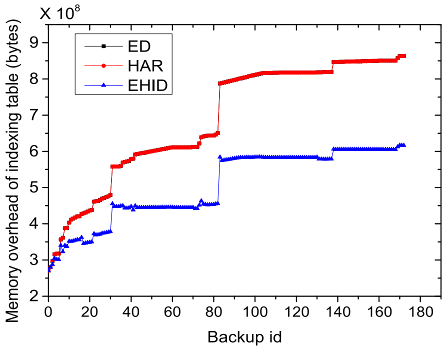
We use the speed factor [19] as a metric to evaluate the restore performance. The speed factor indicates the original data size that can be restored by reading a container from disk on average. A higher value of speed factor indicates better restore performance.

6.2 Evaluating the Memory Overhead of the Indexing Table

We evaluate the memory overhead of the indexing table on Linux and FSL datasets. Please note that the memory overhead means reading all fingerprint entries of the indexing table into memory for completely avoiding disk-based index accesses. In Figure 6(a), the y-axis represents the size of the indexing table that is used to perform deduplication, and the x-axis is the IDs of backup versions. The curves of ED and HAR depicted in Figure 6(a) are completely overlapped. This trend is expected, because these two methods do not separate cold fingerprint entries from the indexing table. EHID can effectively reduce the memory overhead of the indexing table. In addition, with the increase of backup versions, EHID becomes more efficient. For example, the memory overhead of the indexing table incurred by EHID is only 37.80% of the other two methods (ED and HAR) when



(a) Linux dataset



(b) FSL dataset

Fig. 6. Memory overhead of the indexing table.

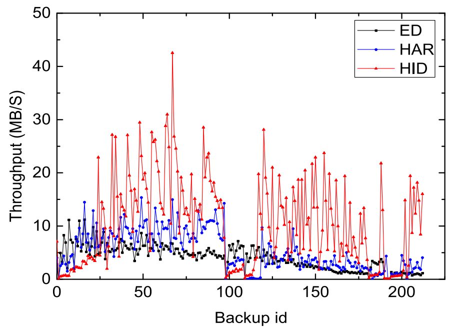
the backup version reaches 213. Figure 6(b) depicts the experimental results when we conduct our experiment on the FSL trace. Figure 6(b) shows similar performance to that on the Linux trace. For instance, the fingerprint entries generated by EHID only account for 71.38% of the other two methods when the backup version reaches 173. These experiments confirm that EHID can effectively reduce the memory overhead of the indexing table and avoid the bloated indexing table caused by system aging.

Figure 6 demonstrates that when using the Linux dataset, EHID can save more memory than that of the FSL dataset. There are two reasons behind this. The first one is that the portion of sparse containers of the Linux dataset is higher than that of the FSL dataset. This is because a large portion of the Linux dataset is self-referenced [10]. The second one is, in contrast to the 173 backup versions of the FSL dataset, that the Linux dataset has 213 backup versions. This is a very important feature in which EHID can save more memory with the growth of backup versions. Additionally, for the Linux dataset, we employ Linux-3.x for the evaluation before backup version 191. The next backup version, 192, uses Linux-4.x. Since the kernel of Linux has a major revision update, the new backup data incurs a large number of unique chunks and corresponding fingerprint entries. This is the reason we observe a spike in Figure 6(a). However, both parts of Figure 6 demonstrate the same trend that with the increase of backup versions, the memory overhead of the indexing table generated by ED and HAR grows significantly. In contrast, the memory overhead of the indexing table incurred by EHID grows much more slowly than that of ED and HAR.

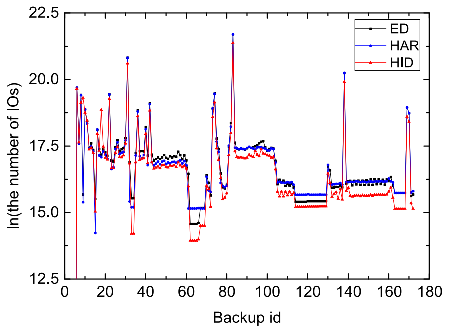
6.3 Evaluating the Backup Performance

We evaluate the backup performance when the Bloom filter is disabled on two datasets (i.e., the Linux and FSL datasets). Figure 7(a) depicts the backup throughput of ED, HAR, and HID when using the Linux dataset. The results reveal that the throughput of HID is the highest one. HAR and ED are ranked second and third, respectively. The reason the throughput of HID with the first 23 backup versions is low is that the inability to cache sparse containers of HID damages the efficiency of distilling hot fingerprint entries.

Since the FSL trace does not contain byte streams of the data chunks, it is infeasible to evaluate the actual throughput on the FSL trace like on the Linux trace. In this part of the study, we investigate the I/O behavior of the methods. Figure 7(b) shows the backup performance of ED, HAR, and HID on the FSL dataset. Due to the large fluctuations in I/O, to make the results more intuitive and understandable, we perform a $ln(I/Os)$ operation on the number of I/Os (the y-axis in Figure 7(b)).

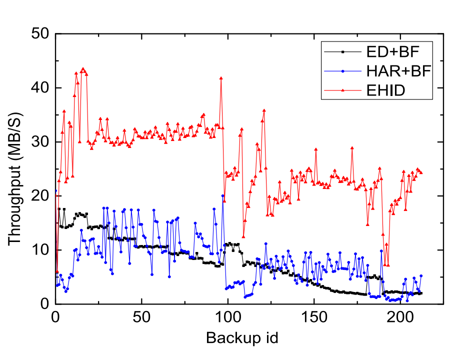


(a) Backup throughput with the Linux dataset

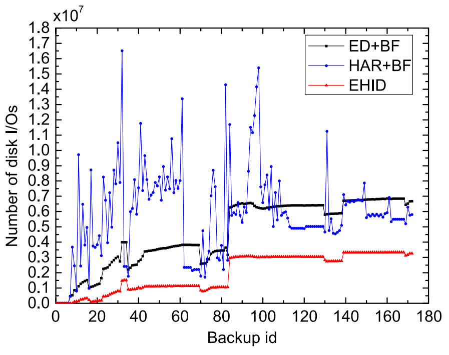


(b) Number of disk I/Os with the FSL dataset

Fig. 7. Backup performance of ED, HAR, and HID.



(a) Backup throughput with the Linux dataset

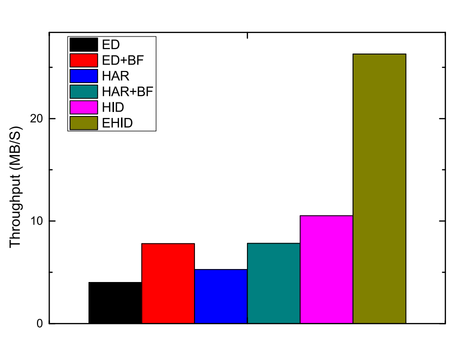


(b) Number of disk I/Os with the FSL dataset

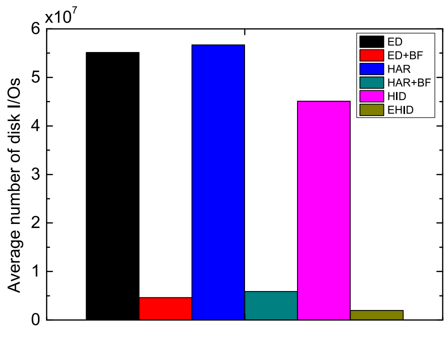
Fig. 8. Backup performance of ED+BF, HAR+BF, and EHID.

Figure 7(b) reveals that compared to the other two methods (i.e., ED and HAR), HID generates the least amount of I/Os.

Now we are in the position to evaluate EHID—an evolved version of HID that integrates a Bloom filter to further boost backup performance. For fair comparison purposes, we configure a Bloom filter for both ED and HAR approaches as well. Figure 8(a) shows the throughput of ED+BF (Bloom filter), HAR+BF, and EHID. The results from Figure 8(a) indicate that the throughput of EHID is much higher than that of the other two methods. There are two factors contributing to EHID's high throughput. First, EHID improves the efficiency of memory and reads more useful fingerprint entries from disk into memory when the same memory size is configured. Second, since all fingerprint entries preserved in memory are hot fingerprint entries, the hit ratio of memory-based index lookups is significantly increased, and thus lookup requests for accessing the disk-based index are noticeably avoided. As expected, the throughput of ED+BF gradually decreases as the backup version increases. This is because the fragmentation of data chunks gradually grows with the increase of the backup versions. Therefore, more lookup requests have to be satisfied with the disk-based index, and such effects result in performance degradation. It is very interesting to observe that the throughput of HAR+BF fluctuates around the curve of ED+BF. The reason behind the throughput fluctuation is that HAR has to update the fingerprint entries of all fragmented chunks, which involves additional disk I/Os. When the number of fragmented chunks

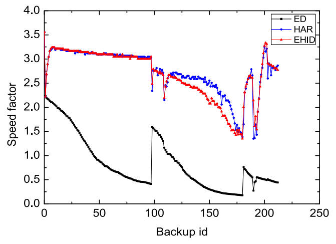


(a) Average backup throughput with the Linux dataset

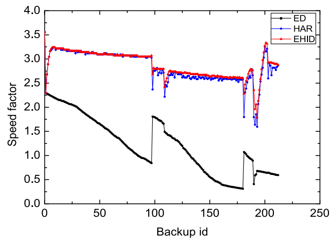


(b) Average disk I/Os with the FSL dataset

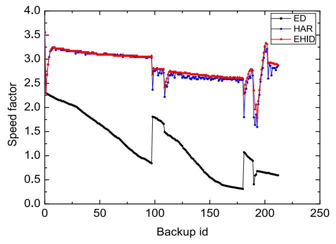
Fig. 9. Average backup performance.



(a) OPT 30



(b) OPT 60



(c) OPT 90

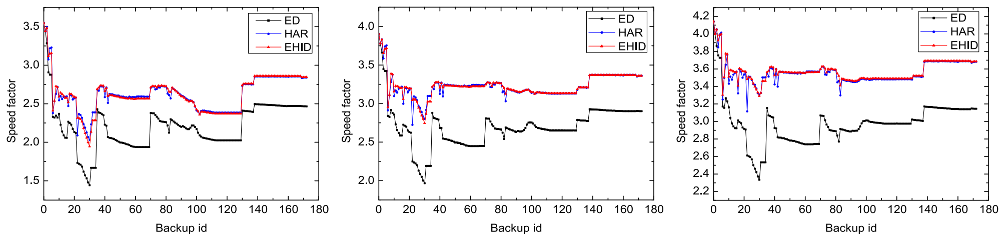
Fig. 10. Speed factor with the Linux dataset.

grows, the throughput of HAR decreases correspondingly due to a large number of generated disk I/Os.

Figure 8(b) demonstrates that the number of disk I/Os issued by EHID for index lookup is significantly lower than that of the other two methods with the Bloom filter enabled when using the FSL dataset. The reason is that EHID significantly alleviates sending lookup requests for the disk-based index. The disk I/Os generated by ED+BF and HAR+BF illustrate a similar behavior on the FSL dataset to that on the Linux dataset. Note that all of these three methods greatly benefit from the Bloom filter, because the Bloom filter effectively avoids accessing the disk-based index incurred by identifying unique chunks. Nevertheless, EHID benefits the most. The average performance of the preceding six methods is shown in Figure 9, which confirms that EHID is significantly better than the other five methods.

6.4 Evaluating the Restore Performance

Now we pay attention to the restore performance. Figure 10 shows the speed factor when using the Linux dataset. Since HAR uses an **Optimal Cache Replacement (OPT)** algorithm, for the sake of fairness, the three methods ED, HAR, and EHID in our experiments are all configured with the OPT algorithm. OPT's cache size is set to 30, 60, and 90, and the cache unit is a container. It can be seen from Figure 10 that the speed factor of ED is the lowest one. As the backup version increases, the speed factor becomes lower. The reason is that ED does not employ a rewriting algorithm to alleviate the fragmentation. The speed factors of HAR and EHID are very similar. More specifically, when the cache size is 30, the average speed factors of HAR and EHID are 2.75 and 2.68, respectively (see Figure 10(a)). When the backup version grows from 115 to 175, HAR



(a) OPT 30 (b) OPT 60 (c) OPT 90

Fig. 11. Speed factor with the FSL dataset.

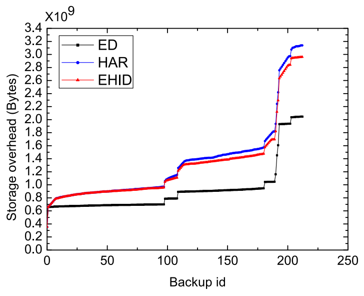
slightly outperforms EHID. This is because HAR rewrites more fragmented chunks than EHID does in these backup versions, which leads to more storage space usage. However, as the cache size increases, the speed factor of EHID becomes better than that of HAR. For example, when the cache size is 60, the average speed factors of HAR and EHID are 2.8327 and 2.8806, respectively. And when the cache size is 90, the average speed factors of HAR and EHID are 2.8328 and 2.8807, respectively.

Additionally, we can see that in both 60- and 90-cache-size cases, EHID outperforms HAR slightly. This change is because EHID only rewrites one fragmented chunk once, whereas HAR may rewrite one fragmented chunk many times. For example, suppose that when there is a fragmented fingerprint entry I1 pointing to a sparse container C0, the fingerprint of a new backup chunk J1 is duplicated with the fingerprint of the fragmented fingerprint entry. The new chunk J1 will be rewritten to a new container C1 to alleviate the data fragmentation. Since a new container is normally not a sparse container, the system should timely update the fragmented fingerprint entry I1 and point it to the container C1. If this fragmented fingerprint entry I1 is not updated before another new chunk J1 is required to back up, the system will rewrite the new chunk J1 to another new container C2. In the original HAR, due to the synchronization of multi-threads and dirty buffer, occasionally the fragmented fingerprint entries may not be able to be updated timely. In doing so, the over-rewritten chunks in HAR waste the storage space and the disk bandwidth when restoring data. In contrast to HAR, EHID is not required to update the fragmented fingerprint entries because all of the fragmented fingerprint entries have been separated from the indexing table. Therefore, EHID does not have the possibility of rewriting the same fragmented chunk many times.

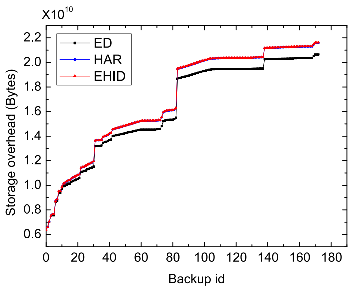
Figure 11 depicts the speed factor when using the FSL dataset. The experimental results are very similar to those of the Linux dataset. As expected, Figure 11 demonstrates that the speed factor of ED is still the lowest one, and the speed factor of EHID is very close to that of HAR. More specifically, as the cache size increases, EHID becomes superior to HAR in terms of the speed factor. For instance, first, when the cache size is 30, the average speed factors of HAR and EHID are 2.6272 and 2.6240, respectively. Next, when the cache size is increased to 60, the average speed factors of HAR and EHID becomes 3.2273 and 3.2311, respectively. Last, when the cache size reaches 90, the average speed factors of HAR and EHID turn into 3.5595 and 3.5738, respectively. It is worth noting that in emergency recovery scenarios, using a larger cache size to restore data is tremendously valuable.

6.5 Evaluating the Storage Overhead

Our experimental results indicate that the average deduplication ratio of the three methods (i.e., ED, HAR, and EHID) ranges from 96.73% to 98.73% on the Linux and FSL datasets. Since the deduplication ratio of these three methods are very similar, to make the performance comparison more



(a) Linux dataset



(b) FSL dataset

Fig. 12. Storage overhead evaluation.

intuitive and clearer, we evaluate the storage overhead instead of the deduplication ratio. Here, the storage overhead refers to the total size of stored chunks accumulated until the deduplication-based backup systems finish the first x (i.e., *backup id* + 1) backups.

Figure 12(a) shows the comparison of the storage overhead between these three methods with the Linux dataset. It can be seen from Figure 12(a) that the disk space used by the ED is the smallest one, and this is because ED does not consider the restore performance and only stores the unique chunks. The storage overhead of HAR is slightly higher than that of EHID, and this is because HAR has an over-rewriting phenomenon. The phenomenon means that a fragmented chunk is rewritten more than two or more times in a backup. This over-rewriting operation reduces the utilization of containers and generates more sparse containers. As a result of the increased number of sparse containers, rewriting algorithms rewrite more fragmented chunks and waste storage space. Experimental results from Figure 12(a) demonstrate that the rewriting ratio of EHID, on average, is 2.83% lower than that of HAR with the Linux dataset. This means that when HAR rewrites 10-GB data chunks, EHID can save about 290 MB of storage space.

Figure 12(b) shows the storage overhead of ED, HAR, and EHID when the dataset is FSL. Similarly, the storage overhead of ED is the smallest one. Since the over-rewriting influence of HAR in the FSL dataset case is not as strong as that in the Linux dataset case, the storage overhead of HAR and EHID is very close. For example, in the 173rd version of the backup, the storage overhead of HAR reaches 21,590,404,782 bytes, whereas the storage overhead of EHID reaches 21,617,550,154 bytes. The storage overhead of EHID is only 0.13% higher than that of HAR, which is negligible.

7 CONCLUSION

In this study, we found that fingerprint entries in the index of traditional deduplication-based backup systems can be classified into three categories: useless fingerprint entries, fragmented fingerprint entries, and hot fingerprint entries. Useless fingerprint entries and fragmented fingerprint entries waste memory space and increase disk I/Os. Additionally, separating fragmented fingerprint entries enables deduplication-based backup systems to directly rewrite fragmented chunks, thereby alleviating data fragmentation. To deal with the performance bottleneck caused by cold fingerprint entries (i.e., useless and fragmented fingerprint entries), we first proposed HID, which separates cold fingerprint entries and preserves only hot fingerprint entries in memory. Compared with the state-of-the-art HAR approach, HID improves both memory utilization and backup performance. Thanks to the reduction in data fragmentation, HID improves restore performance as well.

We introduced the FTU feature to compensate for the shortcoming of the Bloom filter. To this end, we proposed an evolved HID strategy—EHID. EHID embeds a Bloom filter that only maps hot fingerprints, which decreases the false-positive rate. Accordingly, EHID exhibits two salient advantages: EHID avoids disk-based index accesses when identifying unique chunks, and EHID decreases the false-positive rate of the integrated Bloom filter, thus further reducing disk accesses. These salient features keep EHID in high-efficiency mode. Furthermore, EHID can effectively improve the throughput of deduplication-based backup systems. The reason is twofold. First, EHID only stores hot fingerprint entries in memory, which can improve the hit ratio of fingerprint lookups. Second, EHID significantly reduces the number of disk I/Os incurred by disk-based index lookups. More specifically, to speed up fingerprint lookups, EHID separates the indexing table and stores them as two files after a backup completes: a hot index file and a cold index file. If a memory miss occurs, EHID will look up the missed fingerprint entry in the disk-based index. Because the disk-based index in EHID only contains the hot index file, which is a tiny part of the entire indexing table, the size of the disk-based index of EHID is much smaller than that of traditional approaches. Therefore, EHID is much faster than traditional approaches in terms of looking up a fingerprint entry in the disk-based index.

We conducted extensive experiments to quantitatively evaluate the performance of EHID. Experimental results and analysis indicate that EHID can significantly improve the performance of deduplication-based backup systems.

Possible directions for future work include combining EHID and near-ED to further explore the performance behavior of EHID.

REFERENCES

[1] FSL. n.d. Traces and Snapshots Public Archive. Retrieved September 13, 2021 from http://tracer.filesystems.org.

[2] Kernel.org. n.d. The Linux Kernel Archives. Retrieved September 13, 2021 from https://www.kernel.org.

[3] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, and Mark Lillibridge. 2009. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proceedings of the IEEE 2009 International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*. IEEE, Los Alamitos, CA.

[4] Heidi Biggar. 2007. *Experiencing Data De-duplication: Improving Efficiency and Reducing Capacity Requirements*. White Paper. Enterprise Strategy Group.

[5] Burton H. Bloom. 1970. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM* 13, 7 (1970), 422–426.

[6] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. 2019. Sliding look-back window assisted data chunk rewriting for improving deduplication restore performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST'19)*.

[7] Zhichao Cao, Hao Wen, Fenggang Wu, and David H. C. Du. 2018. ALACC: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*.

[8] Biplob K. Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC'10)*.

[9] Yuhui Deng. 2011. What is the future of disk drives, death or rebirth? *ACM Computing Surveys* 43, 3 (2011), 1–27.

[10] 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 *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*.

[11] 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 *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*.

[12] John Gants. 2010. Digital universe decade—Are you ready? Retrieved September 19, 2021 from http://viewer.media.bitpipe.com/938044859_264/1287663101_75/Digital-Universe.pdf.

[13] John Gantz and David Reinsel. 2012. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the Far East. *IDC iView: IDC Analyze the Future* 2012 (2012), 1–16.

[14] Fanglu Guo and Petros Efstathopoulos. 2011. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC'11)*.

[15] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. 2017. SmartMD: A high performance deduplication engine with mixed pages. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*.

[16] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. 2010. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy* 8, 6 (2010), 40–47.

[17] 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*. ACM, New York, NY.

[18] Ricardo Koller and Raju Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transaction on Storage* 6, 3 (2010), 1–26.

[19] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*.

[20] 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 *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST'09)*.

[21] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. 2014. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Transaction on Storage* 10, 2 (2014), 1–22.

[22] Dirk Meister, Jürgen Kaiser, and André Brinkmann. 2013. Block locality caching for data deduplication. In *Proceedings of the 6th International Systems and Storage Conference*. ACM, New York, NY.

[23] Athicha Muthitacharoen, Benjie Chen, and David Mazières. 2001. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP'01)*.

[24] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David H. C. Du. 2011. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proceedings of the 13th IEEE International Conference on High Performance Computing and Communication (HPCC'11)*.

[25] Young Jin Nam, Dongchul Park, and David H. C. Du. 2012. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proceedings of the IEEE 20th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, Los Alamitos, CA.

[26] Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging duplicate locality to accelerate chunking in CDC-based deduplication systems. In *Proceedings of the ACM Symposium on Cloud Computing, (SoCC'19)*. ACM, New York, NY, 220–232.

[27] João Paulo and José Pereira. 2014. A survey and classification of storage deduplication systems. *ACM Computing Surveys* 47, 1 (2014), Article 11, 30 pages.

[28] Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival storage. In *Proceedings of the 1st Conference on File and Storage Technologies (FAST'20)*.

[29] Michael O. Rabin. 1981. *Fingerprinting by Random Polynomials*. Technical Report. Hebrew University of Jerusalem.

[30] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC'12)*.

[31] Dan Feng Wen Xia, Hong Jiang, and Yu Hua. 2011. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *Proceedings of the 2011 USENIX Annual Technical Conference (ATC'11)*.

[32] Nai Xia, Chen Tian, Yan Luo, Hang Liu, and Xiaoliang Wang. 2018. UKSM: Swift memory deduplication via hierarchical and adaptive memory region distilling. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*.

[33] Yongtao Zhou, Yuhui Deng, Laurence T. Yang, Ru Yang, and Lei Si. 2018. LDFS: A low latency in-line data deduplication file system. *IEEE Access* 6 (2018), 15743–15753.

[34] Benjamin Zhu, Kai Li, and R. Hugo Patterson. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08)*.

[35] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Wang Xuan. 2021. The dilemma between deduplication and locality: Can both be achieved? In *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST'21)*.

Received October 2020; revised January 2021; accepted March 2021