An Efficient Delta Compression Framework Seamlessly Integrated into Inline Deduplication

YUCHENG ZHANG, School of Mathematics and Computer Sciences, Nanchang University, Nanchang, China
WENBIN ZENG, School of Mathematics and Computer Sciences, Nanchang University, Nanchang, China
HONG JIANG, Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, United States
DAN FENG, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China
ZICHEN XU, School of Mathematics and Computer Sciences, Nanchang University, Nanchang, China
SHUIBING HE, College of Computer Science and Technology, Zhejiang University, Hangzhou, China
MINGZHE ZHANG, Ant Group, Beijing, China
DAN WU, Nanchang University, Nanchang, China

Delta compression can complement data deduplication by further minimizing redundancy through the compression of non-duplicate data chunks. When adding delta compression to deduplication-based backup systems, however, two primary challenges arise that degrade performance of inline deduplication. First, extra I/Os are introduced along the critical paths of backup and restoration for retrieving base chunks, slowing the system. Second, rewriting techniques prohibit specific data chunks from serving as base chunks for delta compression to improve restore performance, resulting in a loss of compression efficiency.

In this paper, we introduce LoopDelta, a framework that seamlessly integrates delta compression into inline deduplication for backup storage, addressing the aforementioned challenges by using three techniques: (1) dual-locality-based similarity tracking leverages both logical and physical locality to detect most of the similar chunks, which, due to their locality, can be prefetched by piggybacking on routine operations during deduplication, thereby eliminating extra I/Os during backup; (2) cache-aware filter identifies base chunks requiring extra I/Os during restore and prevents their referencing, thus eliminating extra restore I/Os; and (3)

This research was supported by the National Key Research and Development Program of China under Grant 2023YFB4502100, the National Natural Science Foundation of China under Grants 62262042 and 62172361, the Major Projects of Zhejiang Province under Grant LD24F020012, the Pioneer and Leading Goose R&D Program of Zhejiang Province under Grant 2024SSYS0002, and the Jiangxi Provincial Natural Science Foundation under Grant 20224BAB202017.
Authors' Contact Information: Yucheng Zhang, School of Mathematics and Computer Sciences, Nanchang University, Nanchang, Jiangxi, China; e-mail: zhangyc_hust@126.com; Wenbin Zeng, School of Mathematics and Computer Sciences, Nanchang University, Nanchang, Jiangxi, China; e-mail: wenbinzeng@email.ncu.edu.cn; Hong Jiang, Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX, USA; e-mail: hong.jiang@uta.edu; Dan Feng, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, Hubei, China; e-mail: dfeng@hust.edu.cn; Zichen Xu, School of Mathematics and Computer Sciences, Nanchang University, Nanchang, Jiangxi, China; e-mail: xuz@ncu.edu.cn; Shuibing He (Corresponding author), College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: heshuibing@zju.edu.cn; Mingzhe Zhang, Ant Group, Beijing, Beijing, China; e-mail: smartzmz@gmail.com; Dan Wu, Nanchang University, Nanchang, Jiangxi, China; e-mail: wudan@ncu.edu.cn.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 1553-3077/2025/11-ART31
https://doi.org/10.1145/3721485

inversed delta compression, which reverses the roles of base and target chunks in the traditional delta compression approach, thereby allowing for the delta compression of data chunks that are otherwise prohibited as base chunks due to rewriting techniques. Experiments show that LoopDelta increases the compression ratio by 1.28 to 11.33 times over basic deduplication, without significantly affecting backup throughput, and enhances restore performance by up to 3.57 times.

CCS Concepts: • **Information systems** → **Data compression**; **Deduplication**; • **Computer systems organization** → Embedded systems;

Additional Key Words and Phrases: Delta compression, data deduplication, storage system

ACM Reference Format:
Yucheng Zhang, Wenbin Zeng, Hong Jiang, Dan Feng, Zichen Xu, Shuibing He, Mingzhe Zhang, and Dan Wu. 2025. An Efficient Delta Compression Framework Seamlessly Integrated into Inline Deduplication. *ACM Trans. Storage* 21, 4, Article 31 (November 2025), 30 pages. https://doi.org/10.1145/3721485

1 Introduction

Data backup is one of the most important methods for data protection. Backup workloads often involve a substantial amount of data redundancy. To enhance storage space efficiency, backup systems often employ data deduplication, a technique which segments backup data stream into data chunks and removes duplicate chunks from storage. However, data deduplication fails to remove redundancy among non-duplicate but similar chunks. Delta compression, conversely, addresses this limitation by eliminating redundancy among similar (though non-duplicate) chunks. Given the complementary nature of these two techniques, delta compression can effectively complement deduplication, leading to a further reduction of redundant data beyond what deduplication alone can achieve [17–19, 33, 44, 45]. In this paper, we focus on adding delta compression to inline deduplication-based backup systems in a seamless manner.

Delta compression requires base chunks in both the encoding and the decoding processes. For instance, if chunk A_2 is similar to chunk A_1 (the base chunk), the delta compression approach encodes A_2 relative to A_1 and produces a delta file containing the contents present in A_2 but absent in A_1. When A_2 is needed, the delta file is decoded along with the base chunk A_1 to reconstruct A_2. In backup systems, this technique requires extra I/O operations to retrieve base chunks from storage. **Hard drive disks (HDDs)**, despite their poor I/O performance, are frequently selected as the storage medium for backup systems due to their cost-effectiveness. In this paper, we focus on HDD-based backup systems. Consequently, backup systems adopting solely deduplication are inherently I/O-intensive. The implementation of post-deduplication delta compression would further exacerbate the I/O bottleneck and significantly reduce system throughput. Therefore, commercial backup systems often adopt deduplication alone for data reduction, without integrating delta compression.

To add delta compression to inline deduplication-based backup systems, it is essential to minimize the extra I/O overhead induced by delta compression for retrieving base chunks during both backup and restore processes. A typical backup system arranges data chunks into containers, with each container housing several hundred to thousand data chunks, and retrieves metadata from containers during deduplication to accelerate duplicate detection. Previous researches on post-deduplication delta compression for tar-format backup datasets containing numerous small files indicates that, if the containers housing the potential base chunks are targeted for deduplication, the base chunks can be retrieved during the backup process without additional I/Os via piggybacking on I/O operations for prefetching metadata [48, 49].

In backup systems, base chunks for delta compression are not immediately accessible and need to be identified. Our investigation into techniques for identifying base chunks in Section 3.1 reveals that there are two techniques with complementary strengths. One technique leverages the logical locality between adjacent backups to detect the most highly similar chunks, while the other exploits the physical locality preserved in containers to identify the most potential similar chunks. Our analyses in Sections 3.1.4 and 3.2.1 further suggest that (1) by leveraging both logical and physical locality, we can synergize the beneficial attributes of these two techniques, and (2) the containers that hold similar chunks, identified through a combination of logical and physical locality, can be retrieved by piggybacking on I/O operations for retrieving metadata, given that most containers housing the potential similar chunks are targeted for deduplication.

The I/O overheads for reading base chunks during restore also need to be minimized, as they can degrade restore performance. To reduce I/O overheads for reading base chunks during restore, it is essential to identify which base chunks require extra I/Os based on their container IDs. A container ID is an identifier that can be used to locate the corresponding container. However, our analyses in Section 3.2.2 reveal that when data chunks refer to previously written (old) deltas, it is challenging to obtain the container IDs of the base chunks of these deltas. Existing approaches for obtaining them are either susceptible to **garbage collection (GC)** or require additional I/O operations. Our analyses also suggest that by utilizing metadata retrieved during deduplication, it is possible to predict during backup which old deltas will require additional I/Os for fetching their base chunks during restore.

In addition to introducing extra I/Os, adding delta compression to deduplication-based backup systems may also affect rewriting techniques. Rewriting techniques are often employed by backup systems to identify previously written (old) containers with sparse references and avoid deduplicating against these containers, aiming to reduce chunk fragmentation caused by deduplication. It is possible for similar chunks to occur in sparse-reference containers. Treating these similar chunks as base chunks for delta compression may compromise the effectiveness of rewriting techniques, whereas abandoning delta compression would lead to a decrease in compression efficiency. Our analysis in Section 3.3 indicates that by shifting the focus of delta compression to older chunks, as opposed to targeting the data chunks in the ongoing backup (which is the traditional approach), and creating encoded duplicates of these data chunks, the original unencoded versions can be removed during GC. This approach achieves data reduction comparable to delta compression while maintaining the efficacy of rewriting techniques.

Based on the aforementioned observations and insights, this paper introduces LoopDelta,[1] a novel framework that seamlessly integrates delta compression into inline deduplication for backup storage. By combining the following three techniques, LoopDelta maximizes delta compression for non-duplicate chunks without introducing additional I/Os.

— **Dual-locality Similarity Tracking.** LoopDelta identifies containers that house data chunks and base chunks from the most recent backup to detect most of the similar chunks, including those with high similarity, by leveraging both logical and physical locality. Due to the locality, similar chunks within these containers can be retrieved by piggybacking on routine I/O operations for metadata prefetching during deduplication, thereby eliminating the need for additional I/Os.
— **Cache-aware Filter.** Leveraging recently prefetched metadata during deduplication, LoopDelta identifies old deltas whose base chunks would trigger I/O operations during

[1]https://github.com/good-ncu/LoopDelta

restore. By avoiding reference to such deltas, LoopDelta prevents additional I/Os during restore for retrieving base chunks.

— **Inversed Delta Compression.** Regarding detected similar chunks that cannot serve as base chunks due to the constraint of the rewrite technique, LoopDelta delta-encodes these chunks relative to the data chunks in the ongoing backup, creating encoded duplicates of these data chunks. The removal of unencoded data chunks is deferred to the GC process in order to facilitate data reduction. This allows LoopDelta to maximize delta compression without compromising the efficiency of rewriting techniques.

Using real-world datasets, our experimental results demonstrate that LoopDelta boosts both the compression ratio and restore efficiency, adding further benefits to deduplication, while maintaining backup throughput without significant compromise.

2 Background and Related Work

2.1 Data Deduplication

Backup and Restore Processes. Data deduplication is a widely-used data reduction technique. When applied in backup systems, it splits the backup data stream into data chunks, each identified by a fingerprint generated by a secure hash function like SHA1 [11, 28, 30, 32, 43, 47]. These fingerprints are then compared to those of chunks already stored in the system to detect duplicates. Duplicate chunks are referenced to their previously stored versions, eliminating the need for actual data writing and thereby improving storage space efficiency. Unique data chunks, on the other hand, are grouped into larger containers and saved to HDDs. Once the backup process is complete, a *recipe* is recorded, detailing the fingerprint sequence of the backup data stream for easier restoration in the future [15].

The restoration process entails reading data chunks from HDDs, sequentially substituting fingerprints in the recipe with their corresponding data chunks, and ultimately reconstructing the original backup file. During this restoration, the read unit is a container, meaning that to retrieve a specific data chunk, the entire container holding that chunk is loaded into memory [6]. This facilitates the efficient retrieval of multiple data chunks stored within the same container.

Redundancy Locality. Backup tasks generally involve a sequence of replicas of the primary data, where each replica frequently undergoes modifications from a previous backup [2, 26, 38, 40]. *Redundancy locality*, referring to repeating patterns of redundant data among backups, is often exploited to address performance bottlenecks in backup systems that employ data reduction techniques [9, 16, 53].

Redundancy locality can be further categorized into two types: logical locality and physical locality (also known as spatial locality). Logical locality refers to the repeating pattern of duplicate chunks before deduplication, preserved within the recipe and the sequence of consecutive data chunks in the backup data stream. On the other hand, physical locality refers to the pattern of duplicate chunks after deduplication, preserved within containers. Both categories of locality have been extensively leveraged to enhance deduplication performance, particularly in terms of deduplication efficiency and index management [8, 16, 23, 25, 41, 53].

Chunk Fragmentation. Chunk fragmentation arises because the deduplication unit (i.e., the data chunk) represents only a portion of the storage unit (i.e., the container). Unique data chunks from each backup are often stored in new containers. Initially, all data chunks in a container are referenced by the backup, but some may lose their reference in later backups if deleted or changed. Consequently, data chunks from later backups become scattered across multiple containers. This phenomenon is referred to as fragmentation, and chunk fragmentation decreases restore performance [1, 13, 20, 22, 27, 56].

Rewriting. Due to fragmentation, some containers may only contain a few referenced data chunks, which are those referenced by the current backup. These containers are referred to as *sparse-reference containers*, and the referenced data chunks within them are called *fragmented chunks*. To alleviate fragmentation, rewriting techniques are proposed. These techniques identify sparse-reference containers and avoid referencing data chunks within them, thereby enabling duplicate and fragmented chunks to be rewritten into new containers alongside unique data chunks, which improves locality [6, 7, 13, 20, 22, 36].

Among various rewriting strategies, Capping [22] partitions the backup data stream into distinct, non-overlapping segments, where each segment comprises a continuous sequence of data chunks. It restricts the number of old containers (denoted as T) that each segment can deduplicate against. T, also referred to as the capping level, is adjustable. Observing that containers sparsely referenced in one backup often remain so in subsequent backups, Fu et al. introduced the **HAR (History-aware Rewriting)** [13, 14], which identifies sparse-reference containers in one backup and identifies fragmented chunks that reference data chunks from these containers in the next backup.

2.2 Post-Deduplication Delta Compression

Applying delta compression to deduplicated chunks requires three additional stages: (1) similarity detection, (2) retrieving the base chunks, and (3) delta encoding.

Similarity Detection. A sketch calculation approach computes several weak hashes, known as a sketch, for each non-duplicate chunk [5, 29, 50, 54]. Two data chunks are considered similar if their sketches match [3, 10, 21]. To efficiently identify similar chunks, a sketch index is required, indexing the sketches of data chunks within the system. Similar chunks in the system can be detected by querying the index. The strategy for sketch indexing significantly affects the efficiency of delta compression, as it determines which data chunks in the system can serve as base chunks. This will be further discussed in Section 3.1.

Retrieving the Base Chunks. The detected similar chunks must be retrieved from storage to serve as base chunks for both delta encoding and delta decoding processes. In HDD-based backup systems, retrieving these base chunks is a performance bottleneck, especially during the backup process. This bottleneck is the primary reason why delta compression cannot be efficiently employed in backup systems. Previous work [33, 34] suggests that the I/O overhead for retrieving base chunks during backup can considerably degrade backup throughput to an unacceptable level.

Existing solutions adopt two methods to reduce I/O overheads for retrieving base chunks during backup. The first approach is improving physical locality. This method aims to minimize the fragmentation, concentrating base chunks within a limited number of containers, thereby decreasing the number of I/O operations required for retrieving. For instance, MeGA [55] employs service-disruptive offline reorganization to eliminate chunk fragmentation in the most recent backup, whereas FaRE [52] abandons both deduplication and delta compression if fragmentation surpasses a pre-defined threshold to improve physical locality. The second method retrieves potential base candidates by piggybacking on I/O operations for metadata prefetching during deduplication [48, 49], which will be discussed in Section 3.2.1.

Typically, reducing I/O overheads for retrieving base chunks during backup simultaneously decreases such overheads during the restore process. However, there are also studies that directly aim to reduce these overheads during restore. For example, Zhang et al. [51] replicate the conditions of the restore cache during backup. For data chunks whose corresponding base chunks are absent from the restore cache, they abandon delta compression. However, their approach fails to identify base chunks that require additional I/Os during restore when a duplicate chunk references an old delta, which will be discussed in Section 3.2.2.

Delta Encoding. Typically, delta encoding tools are copy-based algorithms derived from Lempel-Ziv [24]. These tools employ a byte-wise sliding window to identify repeated strings between target and base chunks. Then, the repeated strings in the target chunk are substituted with *copy* instructions, encoding the target chunk into a delta file. Due to the elimination of duplicate byte sequences common to both chunks, the resulting delta file occupies less space than the original target chunk. Delta encoding is compute-intensive as it needs to calculate and index hash values to align recurring sequences between the target and reference chunks. Some faster delta encoding approaches such as Edelta [42] and Gdelta [35] have been proposed, which leverage the locality among similar chunks and faster rolling hash to reduce computational overhead.

2.3 Garbage Collection

Usually, each backup file is assigned a retention period, and upon the expiration of this period, the corresponding backup file is deleted. GC is then employed to eliminate invalid (unreferenced) data from the system, thereby consolidating free space [4, 13, 16, 37, 56]. Note that backup systems may occasionally write duplicate chunks for higher write performance. If so, these duplicate chunks are removed during the GC process [1, 9].

Typically, the GC process starts by scanning through active backups to identify and flag the valid data chunks, known as live chunks, which are referenced by backups that haven't expired. In cases where a data chunk exists in multiple physical copies, GC selects one instance, often the most recent, as the live chunk. Following this, these live chunks are extracted from containers that also hold invalid data, and new containers are formed to store them. The containers that have been stripped of their live chunks are then freed up for reuse. GC is time-consuming due to the extensive I/O operations involved. Many techniques have been proposed to decrease its execution time [9, 13, 16, 37, 56].

3 Observations and Motivations

3.1 Distribution of Similar Chunks

The sketch indexing strategy determines which data chunks can be identified as base chunks. Existing sketch indexing techniques can be categorized into three types: logical-locality-based sketch indexing, physical-locality-based sketch indexing, and full sketch indexing. We analyze these three sketch indexing strategies to understand the distribution of similar chunks.

3.1.1 Logical-locality-based Sketch Indexing. Logical-locality-based sketch indexing approaches, such as MeGA [55] and HARD [39], create indexes for sketches of data chunks from the most recent backup, as well as the base chunks of its delta-compressed chunks. Essentially, this sketch indexing technique exploits the logical locality inherent in consecutive backups. Intuitively, a backup is typically a modified version of its predecessor. If data chunks from the previous backup have already been delta compressed and are unsuitable to serve as base chunks due to the challenges of delta decoding during inline backup, the base chunks used in their compression can serve as base chunks for the current backup's delta compression.

One advantage of this sketch indexing technique is the remarkable similarity of base chunks. This advantage stems from the fact that the emergence of similar chunk pairs often originates from minor alterations made to the most recent backup. However, a notable limitation of this technique lies in its potential to overlook certain similar chunks. Our observations indicate that data chunks from a given backup can be derived from multiple preceding backups. Such occurrences, for instance, may arise during data rollback procedures. Consequently, similar chunks may appear across different backup versions. Due to the absence of a direct correlation

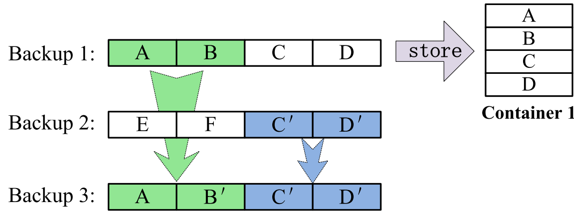


Fig. 1. Chunks B', C', and D' exhibit similarity to the corresponding chunks B, C, and D, respectively. In backup 3, chunks A and B' are derived from backup 1, whereas C' and D' are inherited from backup 2. When attempting to identify similar chunks for the data chunks in backup 3, the logical-locality-based indexing techniques fail to detect chunk B in backup 1 as a similar chunk to B'.

with the most recent backup, these similar chunks may be undetected by the technique leveraging logical locality. Figure 1 provides an example to illustrate how this issue may arise.

3.1.2 Physical-locality-based Sketch Indexing. **Stream-Informed Delta Compression (SIDC)** [34] is a physical-locality-based sketch technique that detects similar chunks from containers deduplicated against. During deduplication, the system determines which data chunks' sketches will be indexed. Specifically, once a container is selected for deduplication, sketches of all data chunks within that container are indexed.

The advantage of this indexing technique lies in its capacity to capture most similar chunks. Logical-locality-based indexing techniques are limited to identifying similar chunks directly related to the last backup. In contrast, the physical-locality-based indexing technique leverages physical locality to overcome this limitation, enabling the detection of similar chunks in backups older than the last backup, as long as they are stored in the same container as a previous copy of a data chunk from the current backup. Taking data chunks in Figure 1 as an example, the physical-locality-based indexing technique can identify the chunk B in Backup 1 as a similar chunk to B' in Backup 3, which cannot be achieved by the logical-locality-based indexing technique. This is because chunk B is stored in Container 1, where a previous version of chunk A also resides.

Within a backup, there may exist both duplicate and similar chunks. The former are termed *self-referenced duplicate chunks*, whereas the latter are known as *self-referenced similar chunks*. When detecting similar chunks for a given data chunk, those detected from previous backups often exhibit higher similarity compared to those detected within the current backup. This may be because the given data chunk could have been derived from similar chunks in previous backups through one or multiple small modifications, whereas no such modification relationship exists between the given data chunk and similar chunks within the current backup. When datasets contain both self-referenced duplicates and self-referenced similar chunks, the physical-locality-based indexing technique may detect low-similarity self-referenced similar chunks as base chunks, thereby compromising the compression ratio. Figure 2 presents an example to illustrate how this problem may arise.

3.1.3 Full Sketch Index. The full sketch indexing technique simply indexes the sketch of every stored data chunk. Since this technique can identify all potential similar chunks, it often serves as an upper bound for compression ratio evaluations when delta compression is involved [39, 51, 55]. Nonetheless, it faces two primary limitations.

Firstly, the size of the indexes scales linearly with the system's storage capacity, posing a challenge in organizing sketch indexes. Storing them on HDDs leads to low query performance, while keeping them in RAM limits the system's scalability. Secondly, this method may result in poor compression when processing datasets containing self-referenced similar chunks, similar to the

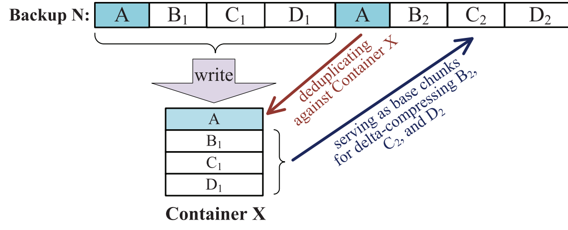


Fig. 2. In backup N, A is a self-referenced duplicate chunk, while B_1, C_1, and D_1 are similar to B_2, C_2, and D_2, respectively. After being processed by the system, the first four data chunks are stored in container X. When processing the fifth chunk (the second A in the backup data stream), the system chooses container X for deduplication, since it contains a duplicate of chunk A. Following this, sketches of the data chunks stored in container X are indexed, and B_1, C_1, and D_1 are detected as the base chunks of B_2, C_2, and D_2.

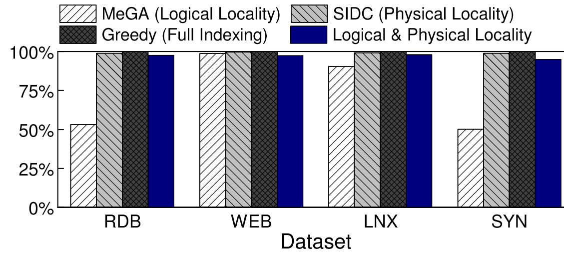


Fig. 3. Percentage of potential similar chunks detected by MeGA, SIDC, Greedy, and the approach exploiting both logical and physical locality on four datasets.

problem faced by physical-locality-based indexing technique. Consider two self-referenced similar chunks, C_1 and C_2. Assume the chunk most similar to C_2, denoted as S, is from a previous backup. If the system processes C_1 first, its sketch may overwrite the sketch of S. This can lead to C_1 being mistakenly detected as the base chunk when processing C_2.

3.1.4 Combining the Best of Both Worlds. The sketch indexing technique that leverages logical locality and the one that exploits physical locality each have distinct advantages in detecting similar chunks, and their strengths complement each other. The technique based on logical locality excels in detecting highly similar chunks, whereas the one relying on physical locality can identify most similar chunks. By combining the strengths of both techniques, we can maximize the benefits of delta compression. Given that data chunks with logical locality are embedded within the data chunks of the most recent backup, and physical locality is preserved in containers, to leverage both logical and physical locality in detecting similar chunks, it suffices to detect similar chunks within containers that include the data chunks of the most recent backup and base chunks of delta-compressed chunks of that backup.

Figures 3 and 4 respectively show the percentage and average **delta compression efficiency (*DCE*)** of detected similar chunks. These metrics are compared between existing sketch indexing techniques and the approach that combines both logical and physical locality, across four datasets. We assume that the full sketch indexing (Greedy) can find all potential similar chunks. The characteristics of these datasets are detailed in Table 2 of Section 6.1. The RDB and SYN datasets contain a significant amount of multi-version inheritance data, while the WEB dataset contains numerous self-referenced duplicates and similar chunks. *DCE*, calculated as $1 - \frac{chunk\ size\ after\ delta\ compression}{chunk\ size\ before\ delta\ compression}$, reflects the similarity of the detected chunks; a higher value denotes greater similarity [50, 54]. The results in the two figures support our analysis, namely,

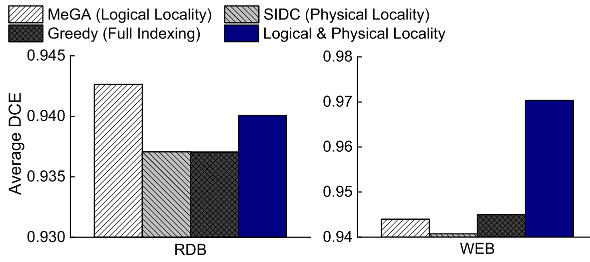
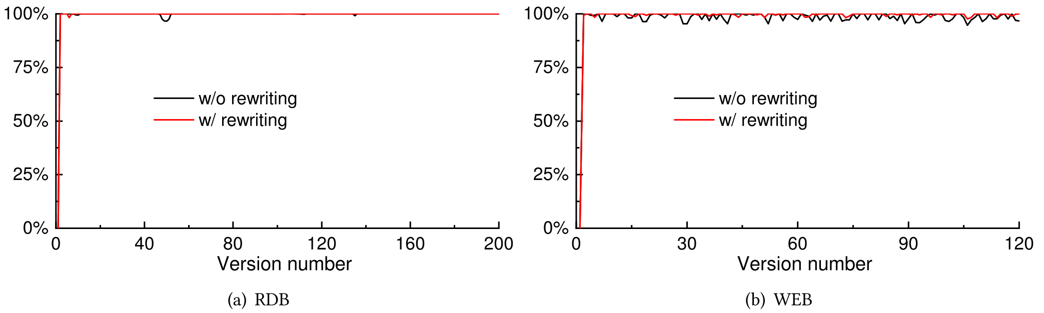


Fig. 4. Average DCE of MeGA, SIDC, Greedy, and the approach exploiting both logical and physical locality on the RDB and WEB datasets.



(a) RDB (b) WEB

Fig. 5. The percentage of containers housing similar chunks, identified by leveraging both logical and physical locality without and with rewriting, targeted for deduplication on the RDB and WEB datasets. The rewriting technique applied is Capping with a capping level of 15 (i.e., each segment references a maximum of 15 old containers).

that combining both logical and physical locality when detecting similar chunks can capture most similar chunks, including highly similar chunks. Notably, on the WEB dataset, the approach combining logical and physical locality achieves a higher *DCE* compared to other methods, as it avoids identifying self-referenced similar chunks as base chunks.

3.2 Avoiding I/Os for Retrieving Base Chunks

The retrieval of base chunks on both the write and read paths requires additional I/Os, which obstructs the use of delta compression in inline deduplication-based backup systems. In this subsection, we explore and evaluate potential strategies to minimize this overhead, ultimately making delta compression a practical and efficient option for inline deduplication-based backup solutions.

3.2.1 On the Write Path. In container-based deduplication systems, such as Data Domain backup systems [53], accessing containers to prefetch metadata for accelerating duplicate detection is a routine operation in data deduplication. This presents an opportunity to minimize I/O overheads for retrieving base chunks. Specifically, when containers storing potential base chunks are targeted for deduplication, these base chunks can be anticipatively retrieved by piggybacking on the routine operations, thereby eliminating the necessity for extra I/O operations. Fortunately, due to redundancy locality, nearly all containers housing similar chunks, identified by exploiting both logical and physical locality, are accessed for metadata prefetching during the deduplication process. This becomes particularly apparent when rewriting is applied, as demonstrated in Figure 5. In other words, most of potential base chunks can be obtained without necessitating extra I/O operations.

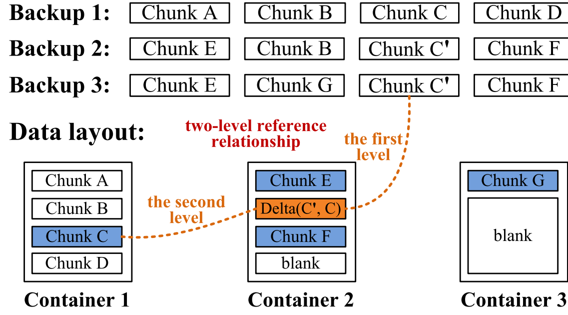


Fig. 6. An example of a base-fragmented chunk. Chunk C' is similar to Chunk C. Delta(C', C) represents a delta encoded Chunk C' using Chunk C as its base. Colored areas in each container represent data chunks required by *backup 3*. Chunk C' in *backup 3* is a base-fragmented chunk because it refers to an old delta, i.e., Delta(C', C), whose base chunk, i.e., Chunk C, requires an extra I/O to fetch Container 1 during the restoration of *backup 3*. The blank space represents the padding content added to a container to ensure it reaches its maximum size (e.g., 4MB) when it is saved to the HDD due to the completion of the backup process.

It is important to note that our claim of eliminating extra I/Os for retrieving base chunks is based on system-level I/O requests, abstracted from the underlying storage details. In a RAID configuration, although the physical I/O operations increase due to the necessity of accessing multiple disks, the strategy of piggybacking base chunk retrieval on metadata prefetching I/Os still avoids incurring additional system-level I/O requests specifically for base chunks. This is achieved by leveraging the parallelism inherent in RAID disks to process the metadata reads, on which the base chunk retrieval is piggybacked. Consequently, despite the increase in physical I/Os, this approach retains its advantage in reducing overall system latency. In what follows, when we refer to 'eliminating extra I/Os for retrieving base chunks', we are specifically referring to system-level I/O requests.

Zhang et al. [48, 49] adopted a similar prefetching strategy for base chunks. However, their strategy is effective only when a significant number of data chunks are repeatedly rewritten in the system. This situation arises primarily when the dataset is in a packed format and contains a large number of small files.

3.2.2 On the Read Path. During the restore process, base chunks also must be loaded into memory to reconstruct data chunks via delta decoding. Detected similar chunks, by using both logical and physical locality, are stored with duplicate chunks (or deltas) that trigger metadata prefetching. This allows for retrieving base chunks during restore without additional I/Os. However, if a duplicate chunk references an old delta, retrieving the base chunk of this delta might require an I/O operation during restore. We call data chunks referring to such deltas *base-fragmented chunks* and suggest rewriting them for better restore speed. Figure 6 showcases a base-fragmented chunk.

In an inline backup system with both deduplication and delta compression, the single level reference relationship is common, such as a duplicate chunk referencing a data chunk, or a delta referencing its base chunk. However, base-fragmented chunks introduce a more complex, two-level reference relationship: a duplicate chunk references a delta, which then references its base chunk, as exemplified by chunk C' in Backup 3 in Figure 6. Establishing each level of the reference relationship typically requires a query, which may require I/Os.

In the aforementioned two-level reference relationship, the first level is established during deduplication by querying the fingerprint index. This index maps the fingerprints of data chunks and

deltas to their container IDs. There are two possible methods to establish the second level of the reference relationship, namely, from deltas to the container IDs of their base chunks. One method is to store base chunk fingerprints alongside deltas and obtain the base chunks' container IDs by querying these fingerprints. This approach requires additional I/O operations. The other approach is to directly store the base chunks' container IDs with the deltas for immediate access. However, this method is vulnerable to GC that can obscure the second level reference, as base chunks may be relocated to other containers during the GC process.

An alternative solution is to rewrite all duplicate chunks referencing old deltas, regardless of whether they are base-fragmented chunks or not. However, our analysis reveals that this approach would lead to a considerable increase in I/Os for data writing, ultimately reducing backup throughput. This is because some datasets contain a large number of duplicate chunks that reference old deltas, while most of these chunks are non-base-fragmented and thus do not need to be written to the HDD. Taking the RDB dataset as an example, when the rewriting approach is Capping with a capping level of 10, 36% of duplicate chunks reference old deltas. Among these 36% of duplicate chunks, 59.9% are non-base-fragmented chunks, which is five times the number of unique data chunks written to the HDD, and hence they are bound to adversely affect backup throughput.

The fact is that, as we found out, the sequence of data chunk processing for both the backup and restore processes is identical. Specifically, during restoration, the order of processing data chunks follows the sequence of fingerprints recorded in the recipe, which corresponds to the order of data chunk processing during backup. Additionally, both the routine metadata prefetching during backup and the prefetching operations for containers holding the required data chunks during restore are executed on a per-container basis, with the only difference being that the former prefetches metadata while the latter prefetches the entire container. Consequently, it becomes possible to identify during the backup process whether the base chunks of old deltas, referenced by a specific backup, will require I/O operations during restore, with the help of the metadata prefetched by routine operations.

3.3 Rewriting-Aligned Delta Compression

The rewriting technique reduces chunk fragmentation by avoiding references to sparse-reference containers. This creates duplicate chunks in the system. During GC, only the most recent version of a duplicate chunk is preserved while older versions are deleted to save storage space. Essentially, rewriting shifts the removal of duplicate chunks from the current backup to previous backups, thereby transferring fragmentation to those earlier backups.

To keep rewriting effective, chunks from sparse-reference containers cannot be used as base chunks for delta compression, which can lead to compression loss. Delta compression involves two steps: first, creating a delta for the target chunk, resulting in encoded and unencoded versions. Second, the unencoded version is deleted. Inspired by the rewriting technique, we discovered that by applying delta compression to old chunks, in a manner that can be described as inversed delta compression compared to traditional methods, rather than to current ones as conventionally done, and by deleting unencoded previous chunks during GC, we can preserve the effectiveness of rewriting while benefiting from delta compression, even for chunks from sparse-reference containers. Figure 7 illustrates an instance of inversed delta compression to exemplify this concept.

4 The Design of LoopDelta

4.1 LoopDelta Overview

LoopDelta is a framework designed to embed delta compression into a typical deduplication strategy that arranges data chunks into containers and prefetches container metadata to accelerate

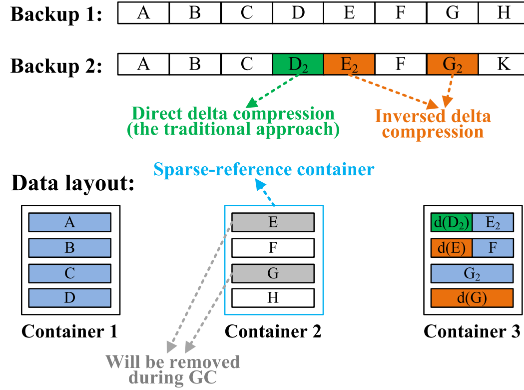


Fig. 7. An example of inversed delta compression. Chunks D_2, E_2, and G_2 in *backup 2* are similar to Chunks D, E, and G, in *backup 1*. After *backup 1* is ingested, its chunks are stored in containers 1 and 2. For *backup 2*, container 2 is a sparse-reference container because it contains only one referenced chunk (i.e., F). Consequently, the system rewrites F in *backup 2* due to its reference to a chunk in container 2, and performs inversed delta compression for E_2 and G_2 because their base chunks E and G are stored in container 2. Here, $d(D_2)$ denotes the delta generated by delta-compressing D_2 relative to D, while $d(E)$ and $d(G)$ represent deltas generated by delta-compressing E and G relative to E_2 and G_2, respectively. Since *backup 2* does not reference any chunk in container 2, thereby preserving the effectiveness of the rewritten of F.

duplicate detection. It employs the following three key techniques to maximize data reduction from delta compression while minimizing I/O overhead for retrieving base chunks.

- **Dual-locality-based Similarity Tracking.** Based on the insights in Section 3.1, LoopDelta tracks containers housing related data chunks from the most recent backup to detect the most similar chunks, including those with high similarity, as detailed in Section 4.2. Because of redundancy locality, these containers are prioritized for deduplication, allowing data chunks within them to be retrieved by piggybacking on routine operations during deduplication to serve as base chunks. This avoids extra I/Os for retrieving base chunks during backup.
- **Cache-aware Filter.** By leveraging recently prefetched metadata from routine operations during deduplication, LoopDelta identifies and rewrites base-fragmented chunks, as detailed in Section 4.3. This eliminates extra I/Os for retrieving base chunks during restore.
- **Inversed Delta Compression.** When similar chunks are detected within sparse-reference containers for specific data chunks, LoopDelta delta-encodes those chunks using these data chunks as the base, thereby creating encoded duplicates of the identified chunks. The removal of unencoded data chunks, a step that contributes to data reduction, is deferred to the GC process, as detailed in Section 4.4. This allows us to reap the benefits of delta compression without compromising the efficiency of rewriting techniques.

The overall workflow of LoopDelta is depicted in Figure 8, which comprises four primary stages that seamlessly integrate delta compression into inline deduplication. In stage (1), the backup workload undergoes chunking and fingerprinting. Stage (2) identifies duplicate chunks via fingerprint indexing. Concurrently, this stage loads potentially similar chunks and their sketches into the potential similar chunk cache. During stage (3), the rewriting technique, if applied, identifies sparse-reference containers and fragmented chunks. Additionally, base-fragmented chunks are identified in stages (2) and (3), as elaborated in Section 4.3. In stage (4), LoopDelta detects similar chunks for unique, fragmented, and base-fragmented chunks within the potential similar chunk cache. If

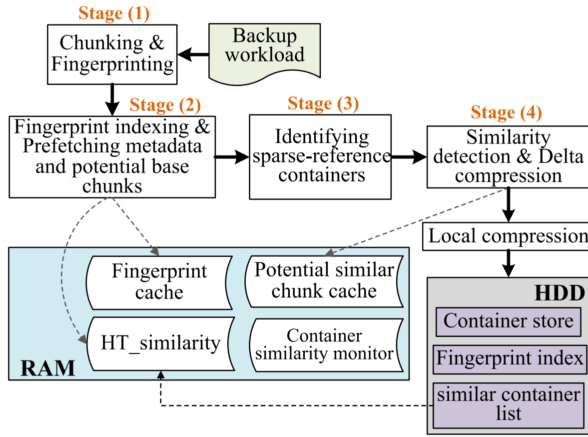


Fig. 8. An overview of LoopDelta. The dashed arrows point to key data structures residing in DRAM required for the corresponding LoopDelta stages.

Table 1. Contents of Data Chunks and Deltas Respectively in Metadata and Data Sections of a Container

Category	Metadata section	Data section
Data chunk	Fingerprint	Chunk contents
		Sketch
Delta	Fingerprint	Delta contents
	Size of its base chunk	
	Fingerprint of its base chunk	

similar chunks are found, delta compression is performed. Ultimately, any unremoved data chunks and deltas are compressed by a local compressor (such as ZSTD [12]) and appended to an open container in memory.

In LoopDelta, a container consists of a metadata section and a data section, the same as that in [16, 53]. The information stored in these two sections is detailed in Table 1. It is important to note that the content saved for a data chunk and a delta in the metadata section differs. When the container in memory attains its maximum size (e.g., 4MB), it is saved to the HDD, and a fresh container is initialized to accommodate subsequent data. Each container in the system is assigned a unique integer ID, which can be used to locate the container. To facilitate understanding of the LoopDelta process, we first provide descriptions of the key data structures involved.

— **Fingerprint Cache:** A cache that stores metadata prefetched from containers.
— **Potential Similar Chunk Cache:** A cache for data chunks and sketches retrieved from containers.
— **HT_similarity:** A lookup table that lists container IDs included in the similar container list generated by the most recent backup. If these containers are accessed during deduplication their data chunks are prefetched into the Potential Similar Chunk Cache.
— **Fingerprint Index:** An index that maps data chunks in the system to their corresponding container IDs.
— **Similar Container List:** A list that records the IDs of containers whose data chunks are referenced during a backup process. Containers whose IDs are recorded in this list will form the HT_similarity for the next backup.

4.2 Prefetching Metadata and Base Chunks

In this subsection, we first describe the routine operation for prefetching metadata in a container-based backup system during deduplication. Subsequently, we elaborate on how LoopDelta identifies similar chunks, i.e., dual-locality-based similarity tracking. Finally, we describe the method for piggybacking potential base chunks onto routine operations.

Routine Operations for Prefetching Metadata. A container includes a metadata section for storing data chunks' metadata and a data section for storing the actual data chunks. The backup system stores a full fingerprint index on disk and employs a fingerprint cache to accelerate duplicate detection by leveraging the physical locality preserved within containers, along with a Bloom filter to rapidly identify non-duplicate chunks. Specifically, each data chunk's fingerprint is compared to the fingerprint cache during storage. If it is not found, a Bloom filter helps determine if the chunk might exist in the system. If a potential match is indicated, the on-disk fingerprint index is checked, and relevant container metadata is prefetched into the fingerprint cache. Redundancy locality increases the chance of finding subsequent chunk fingerprints in the cache, thereby reducing I/O operations for checking the on-disk index.

Dual-locality-based Similarity Tracking. From the observations in Section 3.1, we learn that detecting similar chunks among the data chunks from the most recent backup and the base chunks of previously delta-compressed chunks from the same backup enables the capture of similar chunks with logical locality. Furthermore, identifying similar chunks among data chunks stored within the same containers as the aforementioned data chunks can capture similar chunks with physical locality. To efficiently capture both types of similar chunks for delta compression, LoopDelta identifies the containers housing data chunks from the latest backup and the base chunks of delta-compressed chunks from that backup.

We evaluate the abundance of similar chunks within a container through a metric called container similarity, calculated as $\frac{The\ total\ size\ of\ referenced\ data}{The\ container\ size}$. The referenced data comprises of referenced data chunks (including data chunks written during the current backup) and base chunks. A higher container similarity indicates a greater abundance of similar chunks. This evaluation method implies that containers with more data from the last backup are likely to contain a higher number of similar chunks, emphasizing the primacy of logical locality over physical locality in detecting similar chunks. This is because analyses in Section 3.1 suggest that by solely leveraging logical locality, a significant portion of similar chunks, including those with high similarity, can be identified.

Prefetching Potential Base Chunks. LoopDelta uses a *container similarity monitor* to assess the similarity of containers referenced during the ongoing backup process. As each data chunk undergoes deduplication and delta compression, its size, container ID, and, if delta-compressed, the size and container ID of its base chunk, are used to update the container similarity monitor. Upon completion of a backup, container IDs recorded by the container similarity monitor are written to a file called *similar container list*. Containers in this similar container list, if chosen for deduplication in the next backup, will have their data chunks prefetched as potential base chunks during deduplication.

Specifically, at the start of a backup, the last backup's similarity container list is loaded to memory, and its container IDs are used to create a lookup table named $HT_similarity$. During deduplication, for each container to be accessed by routine operations, LoopDelta checks whether it exists in $HT_similarity$. If a match is found, LoopDelta further checks whether the container is already present in the potential similar chunk cache. If it exists, only the metadata is prefetched; otherwise, both the data chunks (excluding deltas) and metadata are prefetched. If the container does not exist in $HT_similarity$, only the metadata is prefetched. The prefetched metadata is inserted into the fingerprint cache, whereas the prefetched data chunks are inserted into the

potential similar chunk cache. Note that the containers in the last backup's similarity container list might have been reclaimed by GC, thereby hindering the prefetching of similar chunks. We will discuss how to update the similarity container list in Section 4.5. Once the backup completes, the similarity container list from the last backup is no longer required and is thus deleted.

Reducing Transfer Time. Prefetching metadata alone requires minimal disk I/O. Prefetching data chunks along with metadata, while eliminating the seek time and rotational delay of I/Os for reading base chunks, increases transfer time. We employ several strategies to mitigate this transfer time.

Firstly, we store data chunks and deltas separately within the container's data section. As suggested by [33], only non-delta-compressed data chunks are suitable for use as base chunks, thus eliminating the need to prefetch deltas during deduplication. If data chunks and deltas are mixed together in the data section, the system has to read the entire container, including unneeded deltas, to prefetch data chunks and metadata. To address this issue, we separate data chunks and deltas into two distinct areas within the data section, placing the data chunks closer to the metadata. This design allows LoopDelta to prefetch only the required data chunks and metadata, excluding deltas, thereby reducing transfer time.

Moreover, we adopt an approach different from SIDC for storing sketches within the container to reduce transfer time when only prefetching metadata. Unlike SIDC, which stores sketches alongside other metadata, LoopDelta places the sketch of a data chunk immediately next to the corresponding data chunk in the data section. This design prevents sketches from being prefetched when only metadata is the prefetch target, thus reducing transfer time. Compared to SIDC, this design nearly halves the prefetched data size when prefetching metadata alone.

In addition to redesigning the container, we reduce transfer time by selectively disabling data chunk prefetching for certain containers. A straightforward method would be to set a similarity threshold and exclude containers with similarities falling below this threshold from the container similarity monitor, thus keeping them off the similarity container list [46]. However, unless the container similarity threshold can adjust automatically based on the dataset, which is difficult, it may result in the loss of numerous potential base chunks or the unnecessary prefetching of data chunks from containers with minimal similar chunks. This occurs because the average container similarities differ across datasets. Datasets exhibiting relatively high container similarity require a higher similarity threshold, whereas datasets with relatively low container similarity necessitate a lower similarity threshold.

Rather than relying on a similarity threshold to exclude containers from the container similarity monitor, we utilize a ratio of the total size of removed referenced data to the total size of referenced data. We refer to this ratio as the *similarity cutoff*. We define a similarity cutoff (say, $Cutoff$), calculate the total size (say, TS) of all referenced data, including referenced data chunks and base chunks, and use RM to track the size of referenced data removed from the container similarity monitor. Upon backup completion, LoopDelta continuously removes containers with the lowest similarity from the container similarity monitor and adds their referenced data size to RM until the ratio $\frac{RM}{TS}$ reaches $Cutoff$. Subsequently, the container IDs that remain in the container similarity monitor are recorded in the similar container list. This approach automatically disables data chunk prefetching for containers with the least similar chunks, while precisely controlling the proportion of similar chunks that are removed.

4.3 Cache-aware Filter

The cache-aware filter has been devised to detect base-fragmented chunks and rewrite them, thereby enhancing restore performance. Since the fingerprints of deltas' base chunks are utilized in identifying base-fragmented chunks, the fingerprints of deltas' base chunks along with these

deltas' metadata (including fingerprints, chunk sizes, and offsets of deltas in the data section) are stored in the metadata section. These are then prefetched together into the fingerprint cache during the deduplication process. Since the restore process follows the same sequential access pattern as the backup process, the states of the fingerprint cache and restore cache would naturally align if rewriting is not applied. In this scenario, a base-fragmented chunk, which would cause a restore cache miss and trigger an I/O operation during restoration, would not have its fingerprint present in the fingerprint cache during duplicate detection. In other words, without rewriting, base-fragmented chunks can be identified during the duplicate detection phase.

However, the introduction of rewriting may cause inconsistencies between the states of the fingerprint cache and the restore cache. Specifically, containers whose metadata has been prefetched into the fingerprint cache may be classified as sparsely referenced, thereby bypassing prefetching into the restore cache. Consequently, when rewriting is applied, the cache-aware filter employs a two-step process to pinpoint base-fragmented chunks. Initially, it pinpoints base-fragmented chunks that reference deltas missing their corresponding base chunks in the fingerprint cache during stage (2). Subsequently, it detects base-fragmented chunks referencing deltas with base chunks stored in sparse-reference containers.

Specifically, during the duplicate detection stage, when evaluating a particular data chunk, denoted as CK, which references an old delta identified as $Delta_{old}$, the cache-aware filter checks for the presence of the base chunk's fingerprint within the fingerprint cache. If absent, CK is categorized as a base-fragmented chunk. If present, the determination hinges on whether rewriting is applied. In the absence of rewriting, CK is identified as a duplicate chunk. Note that the fingerprint of $Delta_{old}$ can be directly obtained because it is stored together with the fingerprint of CK in the metadata section and is therefore prefetched into the fingerprint cache along with the fingerprint of CK.

If rewriting is applied, the cache-aware filter cannot immediately determine whether CK is base-fragmented. Instead, it associates the detected container ID (CID_{base}) with CK for later determination. In the stage of identifying sparse-reference containers, i.e., stage (3) in Figure 8, if the rewriting approach classifies CK as a fragmented chunk, further investigation into whether it is base-fragmented becomes unnecessary as it has already been confirmed for rewriting. Alternatively, if CK is non-fragmented, the cache-aware filter checks whether the container with ID CID_{base} is sparsely referenced. If true, CK is identified as a base-fragmented chunk; otherwise, it is considered a duplicate chunk.

4.4 Post-Deduplication Delta Compression

Delta Compression Workflow. LoopDelta attempts to perform delta compression on all data chunks requiring storage, including unique, fragmented, and base-fragmented chunks, in order to minimize the amount of stored data. To achieve this, LoopDelta identifies similar chunks for each of the aforementioned data chunks from the potential similar chunk cache, which holds candidates prefetched during deduplicate detection, and performs delta compression if a match is found. In LoopDelta, there are two types of delta compression: traditional direct delta compression and inversed delta compression. The type of delta compression used depends on whether a similar chunk is found in a sparse-reference container. If so, inversed delta compression is performed, as will be further discussed later. If rewriting is not applied, LoopDelta applies only direct delta compression.

Inversed Delta Compression. Consider a new chunk (say, N) with a similar chunk (say, S) found in backup storage. Direct delta compression encodes N relative to S, producing a delta (say, $Delta_{n,s}$), which is then stored instead of N for immediate data reduction. Conversely, inversed delta compression encodes S relative to N, generating a delta ($Delta_{s,n}$), which is stored alongside N. Since inversed delta compression creates an encoded version of S, the original S becomes

redundant in the sparse-reference container and is subsequently removed in the next GC. Essentially, data reduction via inversed delta compression is deferred until the next GC. Typically, when decoding a delta created through inversed delta compression, there is no need for an extra I/O operation to retrieve the base chunk. This is because the delta (e.g., $Delta_{s,n}$) and the base chunk (e.g., N) are generally stored within the same container. However, there are occasions where GC may redistribute them to separate containers.

In addition to achieving data reduction from data chunks with similar chunks in sparse-reference containers, inversed delta compression offers two additional advantages. Firstly, it preserves chunk locality, just like the rewriting technique. Secondly, it enhances the similarity of detected base chunks for the subsequent backups. Since deltas cannot serve as base chunks, in a sequence of similar chunks from different backup versions, if we apply direct delta compression, the initial data chunk consistently serves as the base chunk for delta-compressing subsequent similar chunks in later backups. However, as the interval between backup versions increases, the similarity between the contained chunks typically decreases. Inversed delta compression allows the most recently written similar chunks to serve as the base chunks, thereby enhancing the similarity of detected base chunks.

However, inversed delta compression also comes with two downsides. Firstly, compared to direct delta compression, it increases I/O overhead during data writing due to the need to store additional data. Secondly, it results in a higher number of duplicate chunks being eliminated during GC, which will be further discussed in Section 4.5. Given that LoopDelta is I/O-intensive, it favors direct delta compression. To achieve this, when detecting similar chunks, it gives precedence to similar chunks in sparse-reference containers.

4.5 Garbage Collection

During the GC process, LoopDelta identifies the most recently written instance of a data chunk with multiple physical instances as the live chunk. In this process, the target chunks for inversed delta compression are eliminated. Furthermore, GC might diminish the effectiveness of potential similar chunk prefetching, as the containers whose IDs are recorded in the similar container list could have been reclaimed during GC. To address this problem, LoopDelta updates the similar container list after each GC process. It is worth noting that there is only one list that needs to be updated for each backup data stream. In comparison to GC, the overhead for updating the similar container lists is negligible.

5 Implementation and Discussion

5.1 Rewriting

LoopDelta supports two rewriting approaches, namely, HAR [13] and Capping [22], in its current implementation.

Implementation of Capping. Capping divides the data chunks of the ongoing backup into segments and limits the maximum number of containers that a segment can deduplicate against. To avoid reducing the rewriting efficiency of Capping, base chunks of data chunks in a segment can only be detected from containers that this segment can deduplicate against. However, containers that are identified as sparsely referenced by Capping and thus cannot be deduplicated against may still be prefetched into the potential similar chunk cache. If these containers contain similar chunks, not performing delta compression on them would result in compression loss.

Since the containers that can be deduplicated against may differ from one segment to another, to address this issue, each segment records the containers that its data chunks can deduplicate against. When detecting similar chunks, the system prioritizes detecting similar chunks

in such containers recorded by their corresponding segments. For similar chunks so detected, it performs direct delta compression. Otherwise, it detects similar chunks in other containers in the potential similar chunk cache and performs inversed delta compression if similar chunks exist.

Implementation of HAR. HAR defines a *container's utilization* for a backup as the fraction of its data chunks and base chunks that are referenced by the backup, calculated as $\frac{The\ total\ size\ of\ referenced\ data\ chunks}{The\ container\ size}$. Containers with a utilization falling below a certain rewriting threshold (e.g., 50%) are considered sparsely referenced. To simplify the system design, we modified the definition of "container's utilization" to be consistent with the definition of *container similarity*, which is also calculated as $\frac{The\ total\ size\ of\ referenced\ data\ chunks\ and\ base\ chunks}{The\ container\ size}$. When HAR is applied, LoopDelta uses two potential similar chunk caches: one for direct delta compression and another for inversed delta compression. The system starts with the cache for direct delta compression when detecting similar chunks. The size of the inversed delta compression cache is set to half of the size of the direct delta compression cache. Note that when rewriting is not applied or the rewriting algorithm is Capping, only one potential similar chunk cache is required.

Upon completion of a backup, sparse-reference containers identified by HAR are stored. At the start of a backup, sparse-reference containers from the last backup are loaded into memory to create a lookup table. For a container whose data chunks and metadata are prefetched during deduplication, the system checks whether its ID exists in this lookup table. If so, data chunks in it are inserted into the potential similar chunk cache for inversed delta compression; otherwise, they are inserted into the potential similar chunk cache for direct delta compression.

Discussion. Rewriting, however, may result in LoopDelta missing some similar chunks. This occurs because rewriting can prevent data sharing among sparse-reference containers and both the current and subsequent backups. As a consequence, similar chunks within these containers cannot serve as base chunks for delta-compressing future backups. While inversed delta compression addresses the issue for the current backup, it leaves the problem unresolved for subsequent backups. Furthermore, rewriting can actually enhance the similarity of detected base chunks. This is because rewritten chunks can serve as base chunks, and compared to older data chunks, they tend to share more redundancy with data chunks in the future backups.

5.2 I/O Bottleneck Analysis

LoopDelta is I/O-intensive, with multiple tasks in its workflow requiring I/O operations, specifically: (1) looking up the fingerprint index, (2) prefetching metadata, (3) prefetching potential similar chunks, (4) updating the fingerprint index, and (5) writing back containers. The performance bottleneck varies among these tasks, depending on the redundancy of the datasets. For datasets with high redundancy, where duplicate chunks are abundant, most I/O operations are concentrated in tasks (1), (2), and (3), thereby making them the performance bottleneck. Conversely, datasets with low redundancy, characterized by more unique or dissimilar chunks, result in increased I/O operations in tasks (4) and (5), making them the performance bottleneck.

Rewriting has an impact on the I/O operations within LoopDelta and, consequently, its backup throughput. Generally, rewriting can enhance backup throughput by reducing the overall I/O overhead, particularly by reducing I/O operations for tasks (1), (2), and (3). However, it also results in more data being stored, which raises I/O operations for tasks (4) and (5). Therefore, for datasets with low redundancy, where the primary bottleneck lies in tasks (4) and (5), a more rigid rewriting setting may potentially decrease backup throughput as it may worsen the bottleneck. Additionally, direct delta compression helps reduce I/O operations during task (5), whereas inverted delta compression increases them.

Table 2. Workload Characteristics of the Tested Datasets

Name	Size	Workload descriptions	Key property
RDB	1080GB	200 backups of the redis key-value store database.	Multi-version inheritance
WEB	330GB	120 days' snapshots of the website: news.sina.com. Snapshots of each day are combined into a tar file.	Self-reference duplicate and similar chunks
LNX	284GB	300 versions of Linux kernel source code. Each version is packaged as a tar file.	
SYN	335GB	180 versions of synthetic datasets generated by simulating file create/delete/modify operations.	Multi-version inheritance

6 Performance Evaluation

6.1 Evaluation Setup

Experimental Platform. Our evaluations were performed on a machine equipped with a 12-core Intel Xeon Silver 4215R CPU, 64GB of DRAM, a 2TB HDD, and an 8TB SSD. The SSD and HDD were utilized to simulate user space and backup space, respectively. Initially, datasets were stored on the SSD, read into DRAM during the experiments, and written to the HDD after undergoing data reduction processing.

System Configurations. In our evaluations of LoopDelta and other techniques, deduplication was configured to employ the Rabin-based chunking algorithm [31]. The minimum, average, and maximum chunk sizes were set to 2KB, 8KB, and 64KB, respectively, for the chunking process. The SHA1 hash function was employed for fingerprinting. The fingerprint cache was configured as a 256-slot LRU cache for storing prefetched metadata. Additionally, a 512-container (2GB) LRU cache was set as the restore cache for data restoration.

For post-deduplication delta compression, Odess [54] was adopted for similarity detection, and Xdelta [24] for delta encoding. After deduplication and delta compression, data chunks and deltas were further compressed using the local compressor ZSTD [12] prior to being written into a container. The container size was fixed at 4MB.

Performance Metrics. Three metrics are employed to evaluate the performance of LoopDelta. *The compression ratio* measures the overall data reduction attained through various compression methods, namely data deduplication, delta compression, and local compression. It is calculated as $\frac{original\ bytes}{post\ compression\ bytes}$. A compression ratio greater than 1 indicates data reduction. *The speed factor* (MB/container-read) is defined as the average data size restored per container read and serves as a metric for evaluating restore performance [6, 7, 22]. Higher speed factors indicate superior restore performance. *The backup throughput* is measured as the throughput from the initial reading of the dataset to its final writing on the HDD. Each experiment was repeated five times to ensure stable and reliable average values for backup throughput. Furthermore, the reported speed factor and backup throughput represent the averages obtained from the last 20 backups.

Evaluated Datasets. In our performance evaluation, we employed four datasets, each comprehensively described in Table 2 along with their distinct features. The datasets encompass a wide range of standard workloads, consisting of snapshots from databases and websites, an open-source coding venture, and a synthetic dataset.

6.2 A Performance Study of LoopDelta

6.2.1 *Similarity Cutoff and Container Design.*

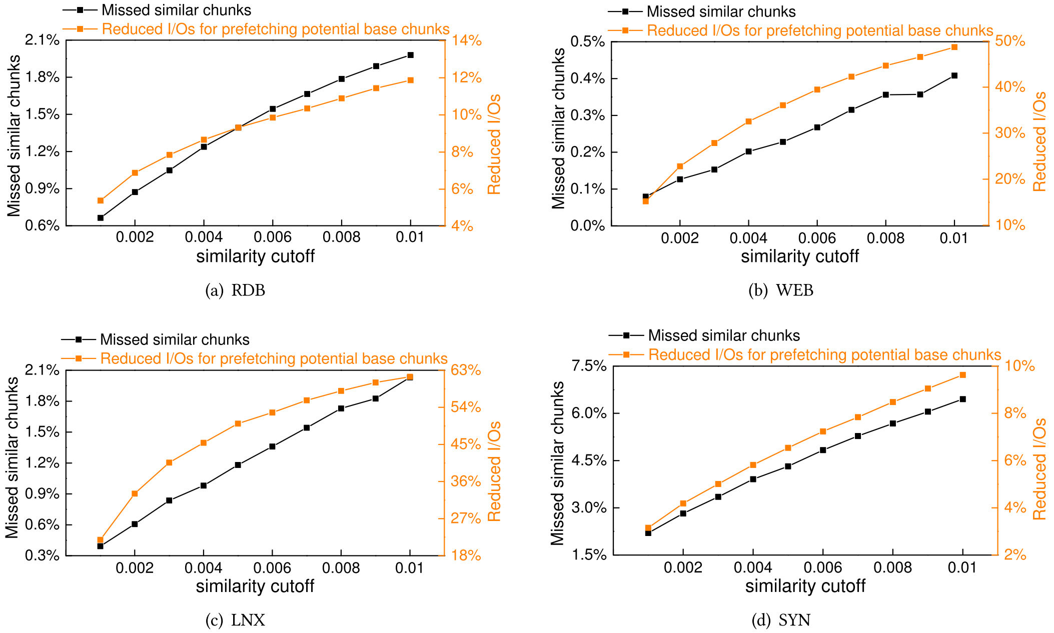


Fig. 9. Percentage of missed similar chunks and the corresponding reduction in I/Os when prefetching data chunks in containers without applying rewriting, as the similarity cutoff varies from 0.001 to 0.01 across four different datasets.

Similarity Cutoff. The similarity cutoff value can influence the number of detected similar chunks, as it prevents containers with minimal similar chunks and the potential base chunk from the most recent backup from being loaded into the potential similar chunk cache. In this subsection, we study LoopDelta's sensitivity to this similarity cutoff. Figure 9 presents the percentage of missed similar chunks and the corresponding reduction in I/O operations when prefetching data chunks in containers without applying rewriting, across four datasets, as the similarity threshold varies from 0.001 to 0.01. Without rewriting, a very small similarity cutoff can effectively reduce I/Os for prefetching potential base chunks. This reduction implies a decrease in transfer time, with only a minimal loss of base chunks for delta compression. Specifically, when the similarity cutoff is set to 0.003, the reduced I/Os for prefetching potential base chunks across the four datasets are 7.8%, 27.9%, 40.6%, and 5%, respectively, while the missed similar chunks are 1%, 0.15%, 0.84%, and 3.3%, respectively.

Furthermore, we also conducted experiments using the approach adopted by [46], which sets a similarity threshold to prevent containers with similarities below this threshold from being loaded into the potential similar chunk cache. However, as Figure 10 demonstrates, this method does not provide relatively precise control over the tradeoff between the percentage of missed potential similar chunks and the reduction in transfer time for prefetching potential base chunks. As an example, with a similarity threshold set to 0.5, the missed similar chunks percentage varies considerably between the LNX and WEB datasets, specifically 66.7% and 2.9%, respectively. The LNX dataset requires a threshold less than 0.1 to minimize missed similar chunks, whereas the WEB dataset can have a threshold set to 0.5. This suggests that there is no one-size-fits-all similarity threshold suitable for all datasets using this approach.

When rewriting is applied, whether it's HAR or Capping, there is no need to set a similarity cutoff to reduce I/Os for prefetching potential base chunks. Firstly, rewriting already significantly

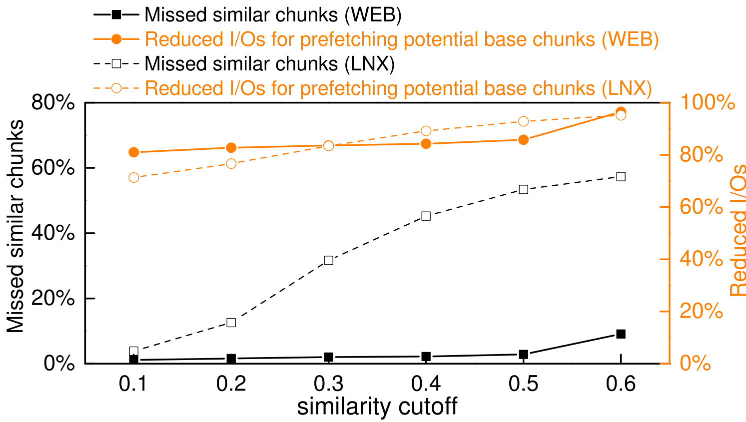


Fig. 10. Percentage of missed similar chunks and the corresponding reduction in I/Os when prefetching data chunks in containers without applying rewriting, as the similarity threshold varies from 0.1 to 0.6 for the WEB and LNX datasets.

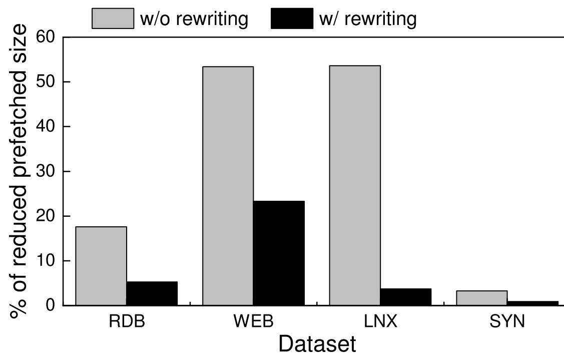
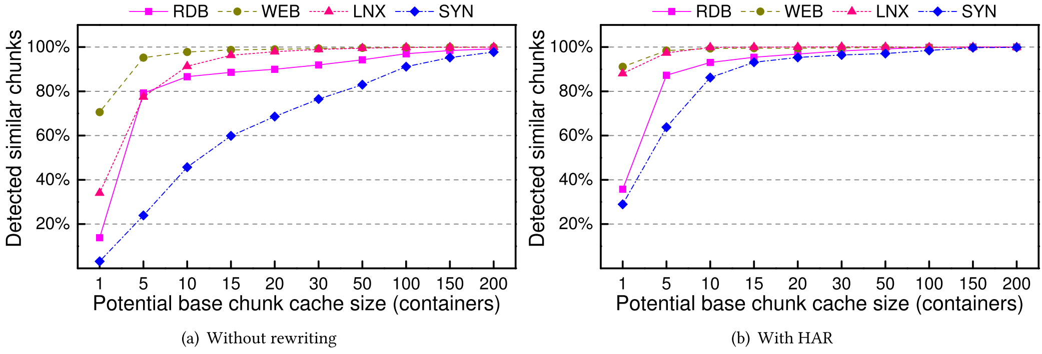


Fig. 11. Percentage reduction in prefetched size resulting from separate storage of data chunks and deltas for LoopDelta without and with rewriting on four datasets. The rewriting technique applied is HAR with a container utilization of 0.4.

reduces I/Os. For example, HAR, with a container utilization of 0.4, decreases I/Os for prefetching potential base chunks by 26.5%, 53.4%, 72.9%, and 54.1% on the RDB, WEB, LNX, and SYN datasets, respectively. Additionally, it reduces total I/Os for prefetching data during deduplication by 23.6%, 28.3%, 75.1%, and 38.5% on these four datasets, respectively. Moreover, rewriting ensures a more uniform distribution of data chunks and the base chunk from the last backup across containers, diminishing the benefits of setting a similarity cutoff to decrease I/Os for prefetching potential base chunks. Attempting to drastically reduce these I/Os would result in the loss of a considerable number of similar chunks. Based on the sensitivity study above, we suggest setting the similarity cutoff to 0.003 when rewriting is not applied, and setting the similarity cutoff to 0 when rewriting is applied.

Container Design. To reduce the transfer time for prefetching potential base chunks, we store data chunks and deltas separately within containers, ensuring that deltas are not read during prefetching. Figure 11 illustrates the percentage reduction in prefetched size attributable to this design for LoopDelta, both without and with rewriting, across four datasets. Without rewriting, our design achieves a reduction in the size of prefetched data ranging from 3.3% to 53.6%. When HAR



(a) Without rewriting

(b) With HAR

Fig. 12. Percentage of detected similar chunks of LoopDelta without rewriting and with HAR as potential similar chunk cache size varies for the four datasets. Each curve shows varying potential similar chunk cache size from left to right: 1, 5, 10, 15, 20, 30, 50, 100, 150, and 200 containers.

is employed, the design reduces the prefetched data size by 0.9% to 23.3%. The effectiveness of this design on the SYN dataset is limited, owing to the limited availability of similar chunks suitable for delta compression within this dataset. Additionally, rewriting diminishes the effectiveness of this design, as it can increase the proportion of data chunks in containers.

6.2.2 Potential Similar Chunk Cache Size. The size of the potential similar chunk cache can affect the number of detected similar chunks. In this subsection, we investigate LoopDelta's sensitivity to the size of the potential similar chunk cache. We evaluate the percentage of similar chunks detected by LoopDelta both without and with rewriting, as the cache size varies from 1 to 200 containers, across four datasets. Figure 12 indicates that, without rewriting, a cache of 200-container size is capable of capturing almost all potential similar chunks. In fact, for the RDB, WEB, and LNX datasets, a cache of just 20-container size can capture more than 90% of the potential similar chunks, whereas for the SYN dataset, a 100-container-sized cache is required. When HAR is applied, a 20-container-sized cache suffices to capture more than 95% of the potential similar chunks for all datasets, including the SYN dataset. Note that, when HAR is applied, the percentage of detected similar chunks with a specific cache size is determined by dividing the number of similar chunks captured with that cache size by the total number of potential similar chunks detectable by LoopDelta with HAR. When the rewriting technique is Capping, the situation is similar to that of HAR. Based on the sensitivity study above, we suggest using a 50-container-sized cache when rewriting is applied and a 200-container-sized cache when no rewriting is used.

*6.2.3 **Cache-Aware Filter (CAF)**.* In this subsection, we assess the efficiency of CAF by comparing the speed factors of three LoopDelta versions: LoopDelta without rewriting, **LoopDelta with Capping (LD-Cap)**, and **LoopDelta with HAR (LD-HAR)**, both with and without CAF. The results are presented in Table 3. According to Table 3, CAF does not significantly enhance restore performance without rewriting, except in the case of the WEB dataset. This exception is attributed to the existence of self-referenced similar and duplicate chunks in the WEB dataset. LoopDelta can only identify similar chunks stored within the same containers as duplicates. If a dataset comprises self-referenced similar and duplicate chunks, a given data chunk might have numerous similar data chunks in the potential similar chunk cache. This situation can lead to inconsistencies in the locality of the detected similar and duplicate chunks. Although this inconsistency has minimal effect on restore performance when the restore cache is adequately

Table 3. Speed Factor Comparison of Six Approaches: LD (LoopDelta) without CAF, LD-Cap without CAF, LD-HAR without CAF, as well as LD with CAF, LD-Cap with CAF, and LD-HAR with CAF

Dataset	RDB	WEB	LNX	SYN
LD w/o CAF	3.01	1.85	1.62	0.86
LD w/ CAF	3.05 (+1.4%)	2.87 (+35.5%)	1.65 (+1.5%)	0.86 (+0.1%)
LD-Cap w/o CAF	1.86	4.48	3.38	1.09
LD-Cap w/ CAF	3.14 (+40.8%)	5.61 (+20.3%)	4.72 (+28.4%)	2.01 (+46%)
LD-HAR w/o CAF	3.21	6.39	6.42	1.24
LD-HAR w/ CAF	4.88 (+34.4%)	7.13 (+10.3%)	9.21 (+30.3%)	1.78 (+30.6%)

Capping's capping level is set to 10, HAR's utilization threshold is set to 0.5.

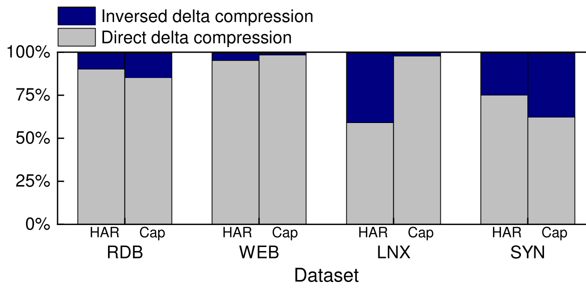


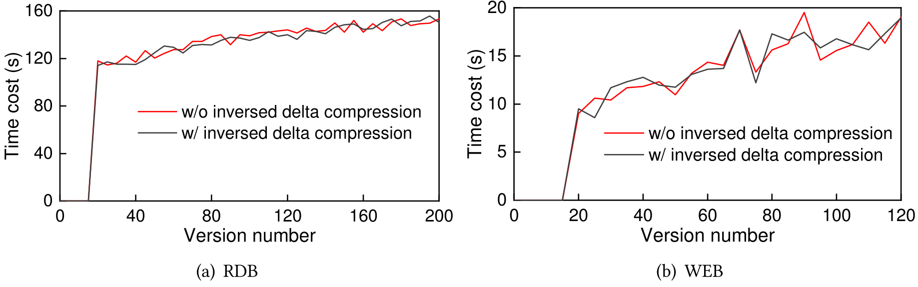
Fig. 13. Proportion of data chunks processed by direct delta compression and inversed delta compression achieved by LoopDelta with HAR and Capping for the four datasets.

sized, it significantly impacts restore performance when duplicate chunks reference old deltas. CAF effectively mitigates this impact and thus improves the restore performance.

When rewriting is applied, CAF considerably improves restore performance, increasing it by 10.3% to 46%. Rewriting efficiently reduces chunk fragmentation introduced by deduplication. However, due to the influence of base-fragmented chunks resulting from delta compression, the improvement in restore performance remains relatively modest. The combination of rewriting and CAF results in a notable enhancement in restore performance.

6.2.4 Inversed Delta Compression. This subsection evaluates the efficiency of inversed delta compression. In the evaluation, the utilization threshold for HAR is set to 50%, while the capping level for Capping is set to 10. Figure 13 indicates that, when the rewriting approach is HAR, between 4.7% and 40.8% of data chunks are processed by inversed delta compression; similarly, when the rewriting approach is Capping, between 1.5% and 37.7% of data chunks undergo inversed delta compression. Since inversed delta compression is capable of obtaining the benefits of delta compression without reducing the efficiency of rewriting, a significant number of data chunks undergoing inversed delta compression ensures LoopDelta's high compression ratio and high restore performance.

However, inversed delta compression leads to more data (extra deltas) being stored. When the rewriting approach is HAR, it results in an increase of 0.56%, 0.54%, 1.38%, and 0.2% in data storage on the RDB, WEB, LNX, and SYN datasets, respectively. When the rewriting approach is Capping, the additional data stored on the four datasets is 0.38%, 0.72%, 0.78%, and 0.31%, respectively. Overall, the additional data required to be written due to inversed delta compression is minimal, as the deltas being written are much smaller in size compared to the data chunks.



(a) RDB (b) WEB

Fig. 14. Time cost of GC for LD-Cap without and with inversed delta compression on the RDB and WEB datasets.

Moreover, inversed delta compression might result in a higher number of data chunks being deduplicated during GC, which could potentially increase the time taken by GC. Figure 14 compares the time cost of GC for LD-Cap, both with and without inversed delta compression, using the RDB and WEB datasets. In this comparison, the capping level for LD-Cap is set to 10. To accumulate more deltas produced by inversed delta compression, GC is performed after every five backups, starting from the 20_{th} backup. As shown in Figure 14, the impact of inversed delta compression on the duration of GC is negligible. This minimal effect stems from the fact that the bottleneck of GC is in marking and advancing the live chunks, a process that demands a considerable amount of I/O operations, rather than in eliminating duplicates.

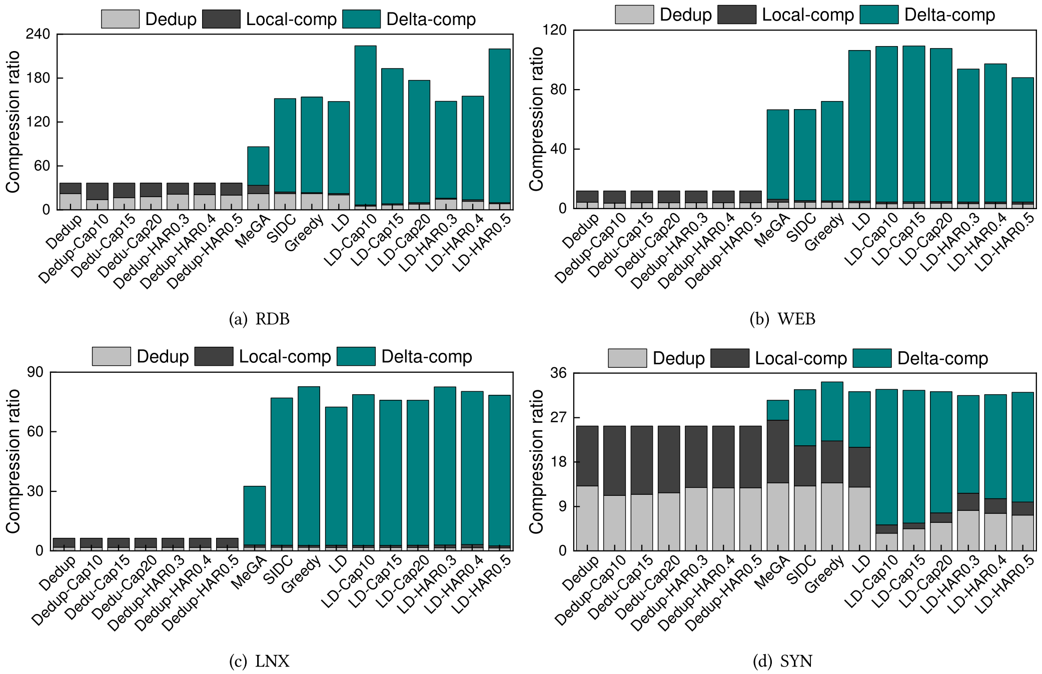
6.3 Comprehensive Evaluation of LoopDelta

In this section, we conduct a thorough evaluation of LoopDelta's performance based on three critical metrics: compression ratio, speed factor, and backup throughput. To facilitate comparison, we also evaluate five other data reduction techniques: Dedup, Dedup-Cap, MeGA, SIDC, and Greedy. Specifically, Dedup, introduced by Zhu et al. [53], is a standard deduplication method that does not involve rewriting. Dedup-Cap and Dedup-HAR refer to Dedup with Capping and HAR, respectively. Similarly, LD, LD-Cap, and LD-HAR refer to basic LoopDelta, LoopDelta with Capping, and LoopDelta with HAR, respectively. Additionally, Dedup-Cap# and LD-Cap# denote Dedup-Cap and LD-Cap with varying capping levels, while Dedup-HAR# and LD-HAR# represent Dedup-HAR and LD-HAR with different utilization thresholds of HAR.

Based on the results and analyses presented in Sections 6.2.1 and 6.2.2, for the evaluations in this section, the similarity cutoff is set to 0.003 and the potential similar chunk cache is set to 200-container-sized for LD; for LD-Cap and LD-HAR, the similarity cutoff is set to 0 and the potential similar chunk cache is set to 50-container-sized.

MeGA, SIDC, and Greedy are three data reduction methods that were elaborated on in Section 3.1. MeGA's restore performance and backup throughput are not compared to LoopDelta due to its use of additional offline and service-disruptive operations that rearrange data chunks and deltas into a non-fragmented layout, significantly improving these metrics and making a comparison unfair. For MeGA, the delta selector threshold is set to 0 to maximize the capture of similar chunks. SIDC's sketch cache size is set to 20MB, the value which they claimed in their paper would achieve the maximum hit rate.

Compression Ratio. Figure 15 indicates that LD obtains a compression ratio comparable to or slightly lower than SIDC and Greedy on the RDB, LNX, and SYN datasets. This suggests that dual-locality-based similarity tracking can capture most of the similar chunks, which can then be prefetched during deduplication to serve as potential similar chunks. On the WEB dataset, LD



(a) RDB

(b) WEB

(c) LNX

(d) SYN

Fig. 15. Comparison of compression ratio achieved by the 17 approaches on the four datasets.

attains the highest compression ratio, which is consistent with our analysis in Section 3.1.4 that LD can detect base chunks with higher similarity than other methods on datasets containing self-referenced similar chunks because it only identifies base chunks from previous backups, thereby avoiding self-referenced similar chunks in the ongoing backup. Specifically, on the WEB dataset, LD achieves a compression ratio that is 1.77×, 1.6×, and 1.48× higher than MeGA, SIDC, and Greedy, respectively. Furthermore, LD also achieves a compression ratio that is 4.05× (RDB), 8.95× (WEB), 11.33× (LNX), and 1.28× (SYN) higher than Dedup.

Another observation from Figure 15 is that rewriting improves LD's compression ratio and even surpasses Greedy's on the RDB dataset. This improvement stems from the fact that rewriting and inversed delta compression increase the *DCE* of base chunks. For instance, on the RDB dataset, the average *DCE* for LD is 0.94, whereas it is 0.979 for LD-Cap10. When the compression ratio is already high, eliminating a small amount of redundancy can lead to a significant boost in the compression ratio. This accounts for why rewriting significantly improves LD's compression ratio on the RDB dataset. However, rewriting can sometimes reduce the compression ratio, as exemplified by the case of HAR on the WEB dataset. This decrease occurs because rewriting may reduce the number of detected similar chunks, as discussed in Section 5.1.

Note that rewriting in LoopDelta can potentially reduce the compression benefits achieved through deduplication. This is because LoopDelta applies delta compression to rewritten data chunks, thereby converting the deduplication gain into a delta compression gain. Furthermore, delta compression may reduce the compression gain obtained from local compression, as there is an overlap in the effectiveness of both delta and local compression methods.

Speed Factor. Figure 16 suggests that LD achieves the highest speed factor among all approaches without rewriting. Post-deduplication delta compression can either decrease or increase the speed factor, depending on the balance between the number of additional I/Os required for retrieving base

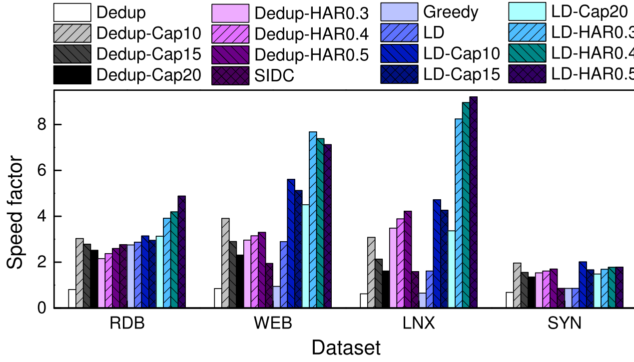


Fig. 16. Comparison of speed factor achieved by the 16 approaches on the four datasets.

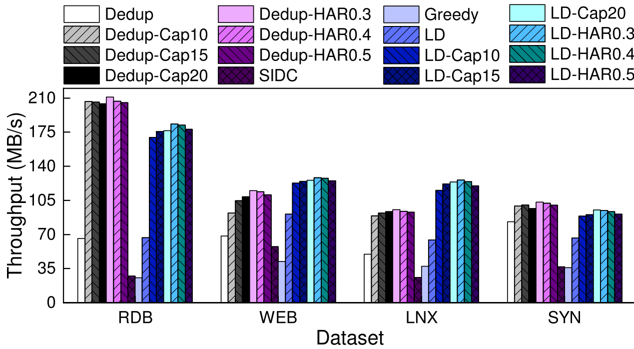


Fig. 17. Comparison of backup throughput achieved by the 16 approaches on the four datasets.

chunks during restore and the reduction in I/Os due to writing less data during backup caused by delta compression. If the former exceeds the latter, the speed factor decreases; conversely, if the latter prevails, the speed factor increases. Since LD achieves a compression ratio comparable to or slightly lower than SIDC and Greedy, without incurring extra I/Os during restore (guaranteed by CAF), its speed factor surpasses not only Dedup, which doesn't use delta compression, but also SIDC and Greedy. On the SYN dataset, the speed factors of SIDC, Greedy, and LD are comparable, primarily due to the limited number of similar chunks in this dataset. Specifically, LD achieves a speed factor that is 3.56× (RDB), 3.4× (WEB), 2.59× (LNX), and 1.25× (SYN) higher than Dedup, 1.04× (RDB), 1.49× (WEB), and 1.02× (LNX) higher than SIDC, and 1.04× (RDB), 3.07× (WEB), and 2.47× (LNX) higher than Greedy.

Rewriting further improves LD's speed factor, resulting in a speedup ranging from 1.03 to 5.7. Additionally, LD-Cap# and LD-HAR# achieve higher speed factors compared to their Dedup counterparts. More specifically, LD-Cap# achieve 1.02-2.09× higher speed factor than Dedup-Cap#, while LD-HAR# attain 1.05-2.59× higher speed factor than Dedup-HAR#.

Backup Throughput. Figure 17 indicates that LD, LD-Cap#, and LD-HAR# exhibit lower backup throughput compared to Dedup, Dedup-Cap#, and Dedup-HAR#, respectively, on the RDB and SYN datasets. The reduction ranges from 1.6% to 20.1%, with an average of 11.9%. An exception is observed where LD achieves comparable backup throughput to Dedup on the RDB dataset. On the WEB and LNX datasets, LD, LD-Cap#, and LD-HAR# demonstrate higher backup throughput than Dedup, Dedup-Cap#, and Dedup-HAR#, respectively, with a speedup ranging from 1.12× to

1.34×. This improvement is attributed to the relatively lower redundancy in these two datasets, where the system bottleneck lies in I/O operations for writing data. Delta compression mitigates this bottleneck.

Furthermore, LD achieves significantly higher backup throughput compared to SIDC and Greedy. Specifically, LD outperforms SIDC by 2.4× (RDB), 1.6× (WEB), 2.5× (LNX), and 1.8× (SYN), and outperforms Greedy by 2.6× (RDB), 2.2× (WEB), 1.7× (LNX), and 1.8× (SYN). Additionally, LD-Cap# and LD-HAR# achieve 2.2× to 6.5× higher backup throughput compared to SIDC and Greedy. Moreover, LD-Cap# and LD-HAR# exhibit higher backup throughput than LD, aligning with the analyses in Section 5.2.

7 Conclusion and Future Work

In this paper, we introduce LoopDelta, a novel framework efficiently integrating delta compression into a typical deduplication-based backup system, which arranges data chunks into containers and prefetches container metadata to accelerate duplicate detection. LoopDelta adopts three techniques to maximize data reduction from delta compression while minimizing I/O overhead for retrieving base chunks. Firstly, dual-locality-based similarity tracking exploits both logical and physical locality to identify most of the similar chunks, which, due to their redundancy locality, can be efficiently retrieved by piggybacking on routine prefetching operations, thus eliminating additional I/O operations during backup. Secondly, cache-aware filter, with the assistance of the fingerprint cache, identifies old deltas whose base chunks that would require additional I/O during restore and prevents referencing these deltas, thereby eliminating extra I/O operations during restore. Thirdly, inversed delta compression encodes similar chunks detected from sparse-reference containers relative to data chunks presented for storage, rather than the traditional reverse approach. This allows us to reap the benefits of delta compression without compromising the efficiency of rewriting techniques. The experimental results demonstrate that LoopDelta significantly enhances the compression ratio and accelerates restore performance compared to deduplication, while minimally impacting backup throughput.

One challenge faced by LoopDelta is how to store the mapping from old container ID to new container ID for chunks during GC when updating the similar container list. For backup systems with large physical capacities, the memory may not be sufficient to store this mapping, and storing it on HDDs would result in slow updating speeds. We plan to address this issue in future work.

References

[1] Yamini Allu, Fred Douglis, Mahesh Kamat, Philip Shilane, Hugo Patterson, and Ben Zhu. 2017. Backup to the future: How workload and hardware changes continually redefine data domain file systems. *Computer* 50, 7 (2017), 64–72.

[2] George Amvrosiadis and Medha Bhadkamkar. 2015. Identifying trends in enterprise data protection systems. In *The 2015 Conference on USENIX Annual Technical Conference (ATC'15)*. USENIX Association, Santa Clara, CA, USA, 151–164.

[3] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. 2009. The design of a similarity based deduplication system. In *The 2nd Annual International Systems and Storage Conference (SYSTOR'09)*. ACM Association, Haifa, Israel, 1–14.

[4] Fabiano C. Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. 2013. Memory efficient sanitization of a deduplicated storage system. In *The 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, San Jose, CA, USA, 81–94.

[5] Andrei Z. Broder. 2000. Identifying and filtering near-duplicate documents. In *Combinatorial Pattern Matching*. Springer, Montreal, Canada, 1–10.

[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 *The 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, USA, 129–142.

[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 *The 16th USENIX Conference on File and Storage Technologies (FAST'18)*. USENIX Association, Oakland, CA, USA, 309–324.

[8] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding up inline storage deduplication using flash memory. In *The 2010 Conference on USENIX Annual Technical Conference (ATC'10)*. USENIX Association, Boston, MA, USA, 1–16.

[9] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. 2017. The logic of physical garbage collection in deduplicating storage. In *The 15th USENIX Conference on File and Storage Technologies (FAST'17)*. USENIX Association, Santa Clara, CA, USA, 29–44.

[10] Fred Douglis and Arun Iyengar. 2003. Application-specific delta-encoding via resemblance detection. In *The 2003 USENIX Conference on USENIX Annual Technical Conference (ATC'03)*. USENIX Association, San Antonio, TX, USA, 113–126.

[11] Kruus Erik, Ungureanu Cristian, and Dubnicki Cezary. 2010. Bimodal content defined chunking for backup streams. In *The 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, San Jose, CA, USA, 1–14.

[12] Facebook. 2024. Zstandard. https://github.com/facebook/zstd. zstd.

[13] Min Fu, Dan Feng, Yu Hua, 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 *The 2014 USENIX Conference on USENIX Annual Technical Conference (ATC'14)*. USENIX Association, Philadelphia, PA, USA, 181–192.

[14] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. 2016. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Transactions on Parallel and Distributed Systems* 27, 3 (2016), 855–868.

[15] 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 *The 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, Santa Clara, CA, USA, 331–345.

[16] Fanglu Guo and Petros Efstathopoulos. 2011. Building a high-performance deduplication system. In *The 2011 USENIX Conference on USENIX Annual Technical Conference (ATC'11)*. USENIX Association, Portland, OR, USA, 1–14.

[17] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. 2008. Difference engine: Harnessing memory redundancy in virtual machines. In *The 5th Symposium on Operating Systems Design and Implementation (OSDI'08)*, USENIX Association, San Diego, CA, USA, 309–322.

[18] Yu Hua, Xue Liu, and Dan Feng. April 27 - May 02, 2014. Neptune: Efficient remote communication services for cloud backups. In *The 33rd IEEE International Conference on Computer Communications (INFOCOM'14)*. IEEE, Toronto, Canada, 844–852.

[19] Navendu Jain, Michael Dahlin, and Renu Tewari. 2005. TAPER: Tiered approach for eliminating redundancy in replica synchronization. In *The 3rd USENIX Conference on File and Storage Technologies (FAST'05)*. USENIX Association, San Francisco, CA, USA, 281–294.

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

[21] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. 2004. Redundancy elimination within large collections of files. In *The 2004 USENIX Annual Technical Conference (ATC'04)*. USENIX Association, Boston, MA, USA, 59–72.

[22] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *The 11th USENIX Conference on File and Storage Technologies (FAST'13)*. USENIX Association, San Jose, CA, USA, 183–197.

[23] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. 2009. Sparse indexing: Large scale, inline deduplication using sampling and locality. In *The 7th USENIX Conference on File and Storage Technologies (FAST'09)*, Vol. 9. USENIX Association, San Jose, CA, USA, 111–123.

[24] Joshua P. MacDonald. 2000. *File System Support for Delta Compression*. University of California, Berkeley, Berkeley, CA.

[25] Dirk Meister, Jürgen Kaiser, and André Brinkmann. 2013. Block locality caching for data deduplication. In *The 6th International Systems and Storage Conference (SYSTOR'13)*. ACM Association, Haifa, Israel, 1–12.

[26] Dutch T. Meyer and William J. Bolosky. 2011. A study of practical deduplication. In *The 9th USENIX Conference on File and Storage Technologies (FAST'11)*. USENIX Association, San Jose, CA, USA, 229–241.

[27] 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 *2011 IEEE 13th International Conference on High Performance Computing and Communications (HPCC'11)*. IEEE Computer Society Press, Banff, Canada, 581–586.

[28] Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging duplicate locality to accelerate chunking in CDC-based deduplication systems. In *The 10th ACM Symposium on Cloud Computing (SoCC'19)*. ACM Association, Santa Cruz, CA, USA, 220–232.

[29] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 2022. DeepSketch: A new machine learning-based reference search technique for post-deduplication delta compression. In *The 20th USENIX Conference on File and Storage Technologies (FAST'22)*. USENIX Association, Santa Clara, CA, USA, 247–264.

[30] Sean Quinlan and Sean Dorward. 2002. Venti: A new approach to archival storage. In *The 1st USENIX Conference on File and Storage Technologies (FAST'02)*. USENIX Association, Monterey, CA, USA, 89–101.

[31] Michael O. Rabin. 1981. *Fingerprinting by Random Polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ.

[32] B. Romański, Ł. Heldt, W. Kilian, K. Lichota, and C. Dubnicki. 2011. Anchor-driven subchunk deduplication. In *The 4th Annual International Systems and Storage Conference (SYSTOR'11)*. ACM Association, Haifa, Israel, 1–13.

[33] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. 2012. WAN optimized replication of backup datasets using stream-informed delta compression. In *The 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, San Jose, CA, USA, 49–63.

[34] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. 2012. Delta compressed and deduplicated storage using stream-informed locality. In *The 4th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'12)*. USENIX Association, Boston, MA, USA.

[35] Haoliang Tan, Wen Xia, Xiangyu Zou, Cai Deng, Qing Liao, and Zhaoquan Gu. 2024. The design of fast delta encoding for delta compression based storage systems. *ACM Transactions on Storage* 20, 4 (Nov. 2024).

[36] Yujuan Tan, Jian Wen, Zhichao Yan, Hong Jiang, Srisa-an Witawas, Baiping Wang, and Hao Luo. 2017. FGDEFRAG: A fine-grained defragmentation approach to improve restore performance. In *The 33rd Symposium on Mass Storage Systems and Technologies (MSST'17)*. IEEE Computer Society Press, Santa Clara, CA.

[37] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2009. Cumulus: Filesystem backup to the cloud. In *The 7th USENIX Conference on File and Storage Technologies (FAST'09)*. USENIX Association, Santa Clara, CA, USA, 225–238.

[38] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. 2012. Characteristics of backup workloads in production systems. In *The 10th USENIX Conference on File and Storage Technologies (FAST'12)*. USENIX Association, San Jose, CA, USA, 1–14.

[39] Chunzhi Wang, Yanlin Fu, Junyi Yan, Xinyun Wu, Yucheng Zhang, Huiling Xia, and Ye Yuan. 2022. A cost-efficient resemblance detection scheme for post-deduplication delta compression in backup systems. *Concurrency and Computation: Practice and Experience* 34, 3 (Feb. 1 2022).

[40] Wen Xia, Hong Jiang, Dan Feng, Fred Douglis, Philip Shilane, Yu Hua, Min Fu, Yucheng Zhang, and Yukun Zhou. 2016. A comprehensive study of the past, present, and future of data deduplication. *Proc. IEEE* 104, 9 (2016), 1681–1710.

[41] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 2011. SiLo: A similarity-locality based near-exact deduplication scheme with low RAM overhead and high throughput. In *The 2011 Conference on USENIX Annual Technical Conference (ATC'11)*. USENIX Association, Portland, OR, 285–298.

[42] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. 2015. Edelta: A word-enlarging based fast delta compression approach. In *The 7th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'15)*. USENIX Association, Santa Clara, CA.

[43] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. 2016. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In *The 2016 Conference on USENIX Annual Technical Conference (ATC'16)*. USENIX Association, Denver, CO, 101–114.

[44] Cui Yong, Lai Zeqi, Wang Xin, Dai Ningwei, and Miao Congcong. 2015. QuickSync: Improving synchronization efficiency for mobile cloud storage services. In *The 21st Annual International Conference on Mobile Computing and Networking (MobiCom'15)*. ACM Association, Paris, France, 592–603.

[45] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. 2005. Deep store: An archival storage system architecture. In *The 21st International Conference on Data Engineering (ICDE'05)*. IEEE, Tokyo, Japan, 804–815.

[46] Yucheng Zhang, Hong Jiang, Dan Feng, Nan Jiang, Taorong Qiu, and Wei Huang. 2023. LoopDelta: Embedding locality-aware opportunistic delta compression in inline deduplication for highly efficient data reduction. In *The 2023 Conference on USENIX Annual Technical Conference (ATC'23)*. USENIX Association, Boston, MA, USA, 133–148.

[47] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. April 26th - May 1st, 2015. AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *The 34th IEEE International Conference on Computer Communications (INFOCOM'15)*. IEEE, Hong Kong, China, 1337–1345.

[48] Yucheng Zhang, Hong Jiang, Mengtian Shi, Chunzhi Wang, Nan Jiang, and Xinyun Wu. 2021. A high-performance post-deduplication delta compression scheme for packed datasets. In *IEEE 39th International Conference on Computer Design (ICCD'21)*. IEEE, 464–471.

[49] Yucheng Zhang, Hong Jiang, Chunzhi Wang, Wei Huang, Meng Chen, Yongxuan Zhang, and Le Zhang. 2024. Applying delta compression to packed datasets for efficient data reduction. *IEEE Trans. Comput.* 73, 1 (Jan. 2024), 73–85.

[50] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 2019. Finesse: Fine-grained feature locality based fast resemblance detection for post-deduplication delta compression. In *The 17th USENIX Conference on File and Storage Technologies (FAST'19)*. USENIX Association, Boston, MA, USA, 121–128.

[51] Yucheng Zhang, Ye Yuan, Dan Feng, Chunzhi Wang, Xinyun Wu, Lingyu Yan, Deng Pan, and Shuanghong Wang. 2020. Improving restore performance for in-line backup system combining deduplication and delta compression. *IEEE Transactions on Parallel and Distributed Systems* 31, 10 (2020), 2302–2314.

[52] Yucheng Zhang, Wenxuan Zhu, Dan Feng, Wei Huang, Nan Jiang, Meng Chen, and Renxin Xia. 2024. A fragmentation-aware redundancy elimination scheme for inline backup systems. *Future Generation Computer Systems-The International Journal of eScience* 156 (Jul. 2024), 53–63.

[53] Benjamin Zhu, Kai Li, and Patterson Hugo. 2008. Avoiding the disk bottleneck in the data domain deduplication file system. In *The 6th USENIX Conference on File and Storage Technologies (FAST'08)*. USENIX Association, San Jose, CA, USA, 269–282.

[54] Xiangyu Zou, Cai Deng, Wen Xia, Philip Shilane, Haoliang Tan, Haijun Zhang, and Xuan Wang. 2021. Odess: Speeding up resemblance detection for redundancy elimination by fast content-defined sampling. In *The 37th International Conference on Data Engineering (ICDE'21)*. IEEE, 480–491.

[55] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. 2022. Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio. In *The 2022 USENIX Annual Technical Conference (ATC'22)*. USENIX Association, Carlsbad, CA, USA, 19–36.

[56] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. 2021. The dilemma between deduplication and locality: Can both be achieved?. In *The 19th USENIX Conference on File and Storage Technologies (FAST'21)*. USENIX Association, 171–185.

Received 30 August 2024; revised 6 December 2024; accepted 24 January 2025