# LoopDelta: Embedding Locality-aware Opportunistic Delta Compression in Inline Deduplication for Highly Efficient Data Reduction

Yucheng Zhang, *School of Mathematics and Computer Sciences, Nanchang University and Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology; Hong Jiang, Department of Computer Science and Engineering, University of Texas at Arlington;* Dan Feng, *Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology; Nan Jiang, School of Information Engineering, East China Jiaotong University;* Taorong Qiu andWei Huang, *School of Mathematics and Computer Sciences, Nanchang University*

https://www.usenix.org/conference/atc23/presentation/zhang-yucheng

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

# LoopDelta: Embedding Locality-aware Opportunistic Delta Compression in Inline Deduplication for Highly Efficient Data Reduction

Yucheng Zhang[†*], Hong Jiang[‡], Dan Feng[*], Nan Jiang[§], Taorong Qiu[†], Wei Huang[†]
[†]School of Mathematics and Computer Sciences, Nanchang University
[‡]Department of Computer Science and Engineering, University of Texas at Arlington
[*]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
[§]School of Information Engineering, East China Jiaotong University
Corresponding author: dfeng@hust.edu.cn

## Abstract

As a complement to data deduplication, delta compression further reduces the data volume by compressing non-duplicate data chunks relative to their similar chunks (base chunks). However, existing post-deduplication delta compression approaches for backup storage either suffer from the low similarity between many detected chunks or miss some potential similar chunks, or suffer from low (backup and restore) throughput due to extra I/Os for reading base chunks or add additional service-disruptive operations to backup systems.

In this paper, we propose LoopDelta to address the above-mentioned problems by an enhanced embedding delta compression scheme in deduplication in a non-intrusive way. The enhanced delta compression scheme combines four key techniques: (1) dual-locality-based similarity tracking to detect potential similar chunks by exploiting both logical and physical locality, (2) locality-aware prefetching to prefetch base chunks to avoid extra I/Os for reading base chunks on the write path, (3) cache-aware filter to avoid extra I/Os for base chunks on the read path, and (4) inversed delta compression to perform delta compression for data chunks that are otherwise forbidden to serve as base chunks by rewriting techniques designed to improve restore performance.

Experimental results indicate that LoopDelta increases the compression ratio by 1.2410.97 times on top of deduplication, without notably affecting the backup throughput, and it improves the restore performance by 1.23.57 times.

## 1 Introduction

Data deduplication has been widely used in computer systems to improve storage space and network bandwidth efficiency [26, 29, 36, 53]. Typically, it removes duplicate data at the chunk granularity (e.g., 8KB size) but fails to eliminate redundancy among (highly) similar but non-duplicate data chunks. Delta compression has been employed to further remove redundant data from post-deduplication non-duplicate but similar chunks, by compressing non-duplicate data chunks relative to their similar chunks (base chunks) [18, 19, 21, 37, 47, 48, 55].

In this paper, we focus on adding delta compression to inline backup storage which usually adopts only deduplication for data reduction. Some efforts have been made to achieve this. Shilane et al. [38] suggested that delta compression achieves more than 2 additional compression on top of deduplication, but it incurs extra I/Os for reading base chunks from storage, which significantly reduce backup throughput. One solution to this problem is to replace the hard disk drive (HDD) with other media with higher random I/O performance, such as solid state drive (SSD). However, this method is not cost-effective because HDD remains significantly cheaper than SSD. So, this paper focuses on HDD-based backup systems.

Zou et al. [55] proposed MeGA to perform delta compression for data chunks whose base chunks can be detected in the last and the current backups on top of deduplication. MeGA is developed based on the assumption that, after each backup, the system will reorganize both the data chunks and the base chunks of the delta-compressed chunks of this backup into a delta-friendly data layout. However, this assumption has a nontrivial impact on both the users and the backup systems, i.e., the backup operations for users are suspended in the reorganization process, and the backup system must perform service-disruptive reorganizations frequently and completes them promptly after each user's backup. This paper focuses on adding delta compression seamlessly to deduplication systems in a non-intrusive and non-service-disruptive manner.

Existing schemes that add delta compression to deduplication systems face many challenges. The first challenge is a low compression ratio. The base chunks required for delta compression are detected by indexing the sketches of chunks, and the compression ratio mainly relies on the strategies adopted to index the sketches [6, 11, 24, 32, 37]. Usually, sketches are weak hashes of the chunk data and can be used to detect simi-lar chunks [4, 6, 11, 21]. Full indexing (indexing the sketches of all data chunks in storage) is the simplest indexing strategy. Our study in Section 3.1 suggests that when similar chunks appear within the same backup, this technique may decrease the compression ratio due to the selection of suboptimal base chunks.

Besides full indexing, the existing sketch indexing techniques can be classified into two categories: the logical-locality-based indexing [42, 55] and the physical-locality-based indexing [37, 38], which have complementary capabilities of detecting similar chunks. The former can detect most of the highly similar chunks by leveraging the logical locality between two adjacent backups but may miss some potential similar chunks; the latter can detect most of the potentially similar chunks by exploiting the physical locality preserved in storage units called containers but suffers from the low similarity of detected chunks. Our analyses in Section 3.1 indicate that the desirable properties of the two techniques can be combined by exploiting both the logical and physical locality.

The second challenge of adding delta compression to deduplication systems is extra I/Os for reading base chunks during backup, which prevent delta compression from being used in high-performance inline backup systems [37, 38]. A typical deduplication-based backup system organizes data chunks into containers each of which consists of hundreds or thousands of data chunks as the storage units [17, 53]. For such a system, a routine operation during deduplication is to prefetch metadata from containers being deduplicated against to accelerate duplicate detection. Our study in Section 3.2 demonstrates that containers holding potential similar chunks detected by using both the logical and physical locality can be prefetched by piggybacking on the routine operations without extra I/Os.

The third challenge is extra I/Os for reading base chunks during restore, which reduce restore performance significantly. More specifically, when delta compression is applied, data chunks may refer to previously written deltas, and the base chunks of these deltas may require extra I/Os during restore. To address this issue, it is necessary to identify such previously written deltas and avoid referring to them. Our analyses in Section 3.2 suggest that existing approaches to identify such deltas either are vulnerable to garbage collection (GC) or require extra I/Os. Our analyses also suggest that the previously written deltas whose base chunks require extra I/Os during restore can be predicted during backup by using the metadata prefetched by the routine operations during deduplication.

The fourth challenge is the potential to miss base chunks when rewriting techniques are applied. Backup systems often adopt rewriting techniques to identify infrequently reused containers and give up deduplicating against them to alleviate chunk fragmentation [14, 16]. To cooperate with rewriting techniques, data chunks in infrequently reused containers cannot serve as base chunks for delta compression, resulting in a compression loss. Our analyses in Section 3.3 suggest that if the target of delta compression is changed to previously written chunks, rather than data chunks in the ongoing backup as in the traditional delta compression method, to generate additional encoded copies of the previously written chunks, the un-encoded copies of previously written chunks can be eliminated during GC to achieve data reduction, which is equivalent to the effect of delta compression.

With the above observations, we propose LoopDelta based on the deduplication strategy that groups data chunks into containers and prefetches metadata from them for duplicate detection during deduplication [17, 53]. By combining the following four key techniques, LoopDelta embeds delta compression in inline deduplication non-intrusively.

Dual-locality Similarity Tracking. LoopDelta tracks data chunks and base chunks of delta-compressed chunks of the immediate predecessor backup to capture highly similar chunks by leveraging the logical locality and tracks the containers holding the aforementioned data chunks to capture similar chunks stored in these containers by exploiting the physical locality.

Locality-aware Prefetching. LoopDelta prefetches base chunks by piggybacking on routine operations to prefetch metadata during deduplication, thereby avoiding extra I/Os for reading base chunks on the write path.

Cache-aware Filter. LoopDelta identifies the previously written deltas whose base chunks require extra I/Os during restore with the assistance of the recently prefetched metadata during deduplication and avoids referring to such deltas, thereby eliminating extra I/Os for reading base chunks on the read path.

Inversed Delta Compression. For data chunks whose detected similar chunks are forbidden to serve as base chunks by rewriting techniques, LoopDelta delta-encodes the detected similar chunks relative to these data chunks while deferring the removal of the data of these delta encoded chunks to the GC process.

Experimental results based on real-world datasets indicate that LoopDelta significantly increases both the compression ratio and restore performance on top of deduplication, without notably affecting backup throughput.

## 2 Background

### 2.1 Data Deduplication

Deduplication and Restore Processes. Typically, a backup stream is divided into data chunks, which are fingerprinted with a secure hash (e.g., SHA1) [12, 31, 33, 35, 46, 49]. Each fingerprint is queried in a fingerprint index to determine whether the system already stores a copy of the fingerprinted data chunk. If true, the system does not store the data chunk but refers it to the previously written copy. Meanwhile, consecutive unique data chunks are grouped into a large I/O unit, called a container, and written to HDDs.

When a backup completes, a recipe recording the fingerprint sequence of the backup stream is stored for the future restoration [16]. When a stored backup is requested, the restore process accesses data chunks one by one according to their order in the recipe to reconstruct the original data. In this process, a restore cache is maintained in memory [7]. The read unit in this process is a container.

Redundancy Locality in Backup Workloads. Backup workloads typically consist of a series of copies of primary data [3,28,41,43]. Redundancy locality in the backup workloads refers to the repeating patterns of the redundant data among consecutive backups [16]. The repeating pattern before deduplication is called logical locality, which is preserved in the recipe and sequence of consecutive data chunks before deduplication. That repeating pattern after deduplication is called physical locality (also called spatial locality [53]), which is preserved in containers.

Both categories of the locality have been widely exploited to improve deduplication performance [9,17,23,27,44,53]. For example, a fingerprint index mapping fingerprints to the physical locations of the chunks is required for detecting duplicates. However, storing the index in HDDs would result in low backup throughput, while putting it in memory would limit system scalability. Zhu et al. [53] put the index in HDDs and alleviate the indexing bottleneck by using physical-locality-based caching. Lillibridge et al. [23] and Guo et al. [17] put the index in memory and reduced its memory footprint by using logical-locality-based sampling and physical-locality-based sampling.

Chunk Fragmentation and Rewriting. Deduplication renders data chunks of a backup stream to be physically scattered, and this is known as chunk fragmentation [14, 22, 30, 56]. Chunk fragmentation decreases the locality and efficiency of the techniques exploiting this locality. For example, it decreases restore performance and backup throughput of container-based backup systems [2, 22]. Backup systems often adopt rewriting approaches to identify infrequently reused containers and give up deduplicating against them to alleviate fragmentation [7,8,15,20,22,39]. Here, the infrequently reused containers are previously written containers that contain only a few data chunks referenced by the current backup. The data chunks that refer to previously written data chunks stored in infrequently reused containers are called fragmented chunks, which will be stored (rewritten) along with unique data chunks to improve the locality of the current backup.

Among rewriting approaches, Capping [22] processes the backup stream in non-overlapping segments, each of which contains a sequence of consecutive data chunks. Within a segment, data chunks can refer to at most T old (previously written) containers. If the number of old containers exceeds T, only the most referenced T containers can be referenced, and data chunks referring to other old containers are rewritten to reduce fragmentation. The value T, also called capping level, is a configurable parameter that trades deduplication for restore performance.

## 2.2 Post-deduplication Delta Compression

Post-deduplication delta compression consists of three stages: (1) resemblance detection (finding similar candidates), (2) reading the base chunks, and (3) delta encoding.

Resemblance Detection. For data chunks not removed by deduplication, a sketch calculation approach calculates sketches for data chunks [32,51,54]. Sketches are usually weak hashes of the chunk data [4, 6, 11, 21]. Two chunks are considered similar if they have the same sketches. The sketch indexing strategy has a critical impact on resemblance detection efficiency, which will be discussed in Section 3.1.

Reading the Base Chunks. Reading the base chunks for an HDD-based system is the performance bottleneck. Shilane et al. [37, 38] indicated that I/O overheads required to read back the base chunks decrease the backup throughput to an unacceptable level. MeGA [55] monitors the containers holding the base chunks and does not perform delta compression for the data chunks whose base chunks are stored in rarely referenced containers to reduce I/Os for reading base chunks. Besides, PFC-delta [52] prefetches base chunks by piggybacking on the routine I/Os during deduplication.

Delta Encoding. Xdelta [25] is a popular delta encoding technique that employs hashing and indexing to identify and eliminate repeated strings between the target and base chunks. Edelta [45] simplifies this process by replacing some of the hashing and indexing operations with fast byte-wise comparisons through exploiting fine-grained locality between similar chunks.

## 2.3 Garbage Collection

Garbage collection (GC) removes invalid chunks (chunks not referenced by any unexpired backups) from the system to consolidate free space [5, 14, 17, 40, 56]. Duplicate chunks will be removed in the GC process [2, 10]. To improve write performance, a deduplication-based backup system might choose to write (rewrite) occasional duplicate chunks while deferring deduplication to a GC process [2].

GC first traverses live backups and marks the live chunks. For a data chunk with multiple physical instances, GC marks one (often the most recently written one) of them as the live chunk [10]. Then, it copies live chunks from partially-invalid containers to form new containers. Then, previous containers whose live chunks are copied out of are reclaimed.

# 3 Observations and Motivations

## 3.1 Analysis of Sketch Indexing Efficiency

Besides the full indexing technique, existing sketch indexing techniques can be divided into two categories: logical-locality-based indexing techniques and physical-locality-based indexing techniques. This section analyzes the efficiency of these two categories of indexing techniques as well as the full indexing technique.

### 3.1.1 Logical-locality-based Sketch Indexing

Substantial similar chunks for delta-compressing a backup can be detected from data chunks of its immediate predecessor backup and similar chunks of this backup. This is because

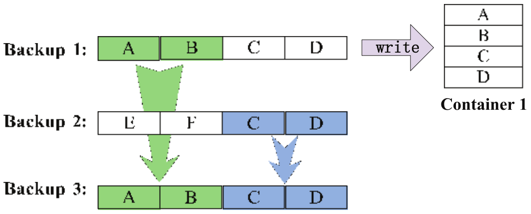


Figure 1: Chunks $B^0$, $C^0$, and $D^0$ are respectively similar to chunks B, C, and D. For backup 3, A and $B^0$ inherit from backup 1, while $C^0$ and $D^0$ inherit from backup 2. When detecting similar chunks for data chunks of backup 3, the logical-locality-based indexing techniques will miss B in backup 1, which is actually similar to $B^0$.

each backup is often a modified copy of the last backup. Based on similar observations, MeGA [55] and HARD [42] index sketches of the data chunks and base chunks of the delta-compressed chunks of the last backup for resemblance detection. These sketch indexing techniques essentially exploit the logical locality between two adjacent backups.

An advantage of these indexing techniques is the high similarity of base chunks. In most cases, the best base chunk for the delta compression of a data chunk is its previous copy in the last backup because similar chunks are usually stemming from small edits to the last backup. A disadvantage of these indexing techniques is that they may miss some potential similar chunks. We observe that data chunks of a backup can inherit from multiple previous backup versions, for example, when data rollback occurs. This means that similar chunks may exist across backup versions. These similar chunks do not have a direct relationship to the immediate predecessor backup and thus cannot be detected by logical-locality-based indexing techniques. Figure 1 gives an example to illustrate how this problem may arise.

### 3.1.2 Physical-locality-based Sketch Indexing

Stream-Informed delta compression (SIDC) [38] is a physical-locality-based sketch technique that detects similar chunks by exploiting the physical locality among backups. Since physical locality is preserved in containers, SIDC is built on container-based deduplication systems. When a duplicate chunk is detected, the container holding the most recently written instance of this duplicate is selected for deduplicating against, and sketches of all data chunks in this container are indexed for matching similar chunks.

An advantage of this indexing technique is that it can detect most similar chunks including the ones missed by logical-locality-based indexing techniques. For data chunks that are inherited from versions older than the immediate predecessor backup, their sketches would be indexed if they are stored in the containers that would be deduplicated against, even if they do not have a direct relationship to the last backup. As shown in Figure 1, when the system deduplicates chunks in backup 3, container 1 is selected for deduplicating against because it holds the previously written copy of chunk A. Therefore, the sketches of B in this container will be indexed and detected as a similar chunk of $B^0$.

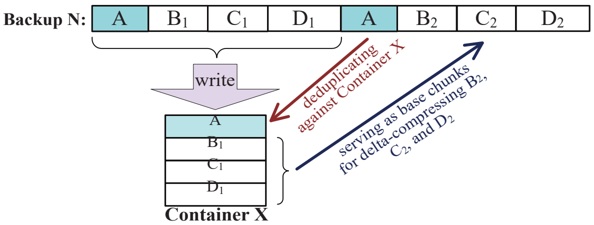

Backup N:
A
B1
C1
D1
A
B2
C2
D2
write
deduplicating against Container X
serving as base chunks for delta-compressing B2, C2, and D2
A
B1
C1
D1
Container X

Figure 2: A in backup N is a self-referenced duplicate chunk, and the rest are self-referenced similar chunks. $B_1$, $C_1$, and $D_1$ are similar to $B_2$, $C_2$, and $D_2$, respectively. The first four data chunks are stored in container X. When deduplicating the second A, the system will select container X for deduplicating against because this container holds the previous-written copy of A. Then, sketches of data chunks in container X are indexed and $B_1$, $C_1$, and $D_1$ are detected as the base chunks for delta-compressing $B_2$, $C_2$, and $D_2$.

A backup may contain duplicates and similar chunks in itself, where the former are referred to as self-referenced duplicate chunks and the latter as self-referenced similar chunks. Compared with self-referenced similar chunks, similar chunks detected from the previous backups tend to share more redundancy with unique chunks of the on-going backup. This is because the latter are much more likely to result from the former after they are slightly modified. Consequently, the physical-locality-based sketch indexing technique can be suboptimal for datasets containing self-referenced duplicate and self-referenced similar chunks. This is because self-referenced duplicate chunks may cause the newly-written containers holding data chunks of the on-going backup to be selected for deduplicating against, which further causes the self-referenced similar chunks to be matched as the base chunks for delta compression. Figure 2 presents a simplified example to illustrate how the problem may arise.

### 3.1.3 Full Sketch Index

The full sketch indexing technique indexes sketches of all data chunks in the backup system and it is often serves as an upper bound for compression ratio evaluations when delta compression is involved [42, 52, 55]. Since the size of sketch indexes grows with the number of backup versions, it is challenging to organize the sketch indexes. Maintaining them in memory would limit system scalability, while putting them in HDDs would greatly reduce query throughput.

Another problem facing this technique is that it can be suboptimal for datasets containing self-referenced similar chunks. This technique indexes sketches of all data chunks in the system, including self-referenced similar chunks. Thus, when self-referenced similar chunks are ingested, the sketches of the best base chunk candidates for delta compression in previous backups may be replaced by those of self-referenced similar chunks, causing self-referenced similar chunks to be

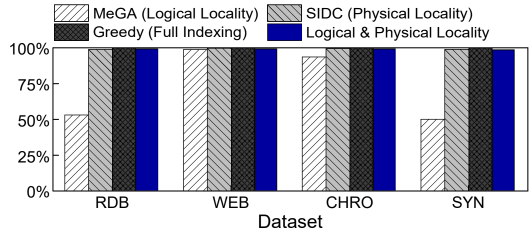


Figure 3: Percentage of potential similar chunks detected by MeGA, SIDC, Greedy, and the approach exploiting both logical and physical locality on four datasets.

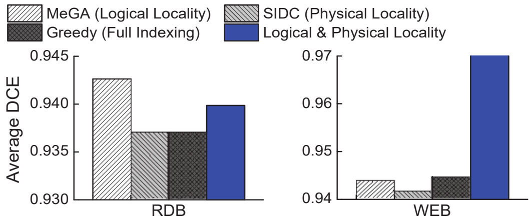


Figure 4: Average DCE of MeGA, SIDC, Greedy, and the approach exploiting both logical and physical locality on the RDB and WEB datasets.

matched as base chunks for delta compression.

### 3.1.4 Combining the Best of Both Worlds

Figures 3 and 4 respectively show the percentage and the average delta compression efficiency (DCE) [51,54] of detected similar chunks of existing sketch indexing techniques including MeGA, SIDC, and the full indexing (Greedy) for datasets whose characteristics are detailed in Table 1 in Section 5.1. Multi-version inheritance is common in RDB and SYN datasets, and the WEB dataset contains substantial self-referenced duplicates and similar chunks. DCE, which is calculated as $1 \frac{\text{chunk size after delta compression}}{\text{chunk size before delta compression}}$, measures the similarity of the detected similar chunks. A larger value of DCE indicates higher similarity.

The results in the two figures agree with the earlier analysis in this section, i.e., the logical-locality-based technique can detect most highly similar chunks but may miss potential similar chunks, while the physical-locality-based technique can detect most potential similar chunks but suffers from low similarity. We found that the two categories of techniques have complementary capabilities for detecting similar chunks. This motivates us to propose a dual-locality-based sketch indexing approach to combine the advantages of both logical and physical locality. The dual-locality-based approach can not only detect most potentially similar chunks but also ensure high similarity between detected chunks, as shown in Figures 3 and 4. Since the full indexing technique can be suboptimal for datasets containing self-referenced similar chunks, the dual-locality-based approach can achieve higher DCE than the full indexing on such datasets.

## 3.2 Avoiding I/Os for Reading Base Chunks

Extra I/O overheads for reading base chunks on both the write and read paths prevent delta compression from being used in high-performance backup systems. In this subsection, we discuss and analyze the possible approaches to reducing or eliminating I/O overheads for reading base chunks to make delta compression feasible and practical for backup systems.

### 3.2.1 On the Write Path

For container-based deduplication systems, such as Data Domain backup systems [53], a routine operation during deduplication is to access containers for prefetching metadata to accelerate duplicate detection, which provides an opportunity to eliminate I/Os for reading base chunks. If the containers holding similar chunks will be accessed for prefetching metadata during deduplication, base chunks can be prefetched by piggybacking on the routine operations during deduplication without requiring extra I/Os.

Fortunately, if similar data chunks are detected by exploiting both logical and physical locality, as suggested in Section 3.1.4, due to redundancy locality, most of the containers holding potential similar data chunks would be prefetched during the metadata prefetching process in data deduplication. As a result, potential similar chunks can be prefetched, i.e., piggybacked on the retrieval of the metadata to serve as potential base chunks, thereby avoiding extra I/Os for reading base chunks on the write path. Zhang et al. [50] employed a similar base-chunk prefetching technique based on different observations, but their approach can only be applied to specific backup datasets (i.e., packed datasets containing substantial small files) and detects similar chunks only when rewriting is applied. In contrast, our approach can be applied to all backup datasets and can work without rewriting.

### 3.2.2 On the Read Path

During restore, base chunks of deltas also need to be read from storage for delta decoding. If base chunks are prefetched along with metadata during deduplication, they would be prefetched along with other chunks (or deltas) during restore, without requiring extra I/Os. However, when a data chunk refers to an old (previously written) delta, the base chunk of this delta may require extra I/Os during restore and decrease restore performance. Data chunks that refer to old deltas whose base chunks trigger read operations during restore are referred to as base-fragmented chunks, which should be rewritten for improved restore performance.

Existing approaches either are vulnerable to GC or require extra I/Os. SDC [52] can identify base-fragmented chunks by simulating the data restore process using container IDs during backup. However, it requires knowledge of the container IDs of the base chunks of referenced old deltas, which is difficult to obtain. Storing container IDs of base chunks along with deltas in containers is vulnerable to GC because base chunks can move around due to GC. Storing fingerprints of base chunks and obtaining their container IDs through the fingerprint index and cache may incur a large number of extra I/Os. Rewriting techniques, which are employed to identify fragmented chunks for deduplication systems, face the same problem in identifying base-fragmented chunks because they also need the container IDs of base chunks for calculating the containers' reuse ratios.

When excluding base chunks, the order in which data chunks are processed during backup is the same as that during restore. Thus, if the base chunk of a referenced old delta can be directly found in the restore cache during restore and thus does not require extra I/O, its fingerprint would also be directly found in the prefetched metadata during deduplication. That is, base-fragmented chunks can be identified with the assistance of the prefetched metadata during deduplication.

## 3.3 Fine-grained Redundancy Prohibited by Rewriting

The rewriting technique declares infrequently reused containers, which should not "share" redundant data with the current backup to alleviate chunk fragmentation. To this end, fragmented chunks need to be rewritten. To cooperate with rewriting, data chunks in the infrequently reused containers also cannot serve as base chunks for delta compression. However, existing rewriting techniques only consider duplicate chunks when identifying infrequently reused containers, without the consideration of similar chunks. In practice, infrequently reused containers may contain many similar chunks. This will lead to a significant compression loss if these similar chunks cannot serve as base chunks for delta compression.

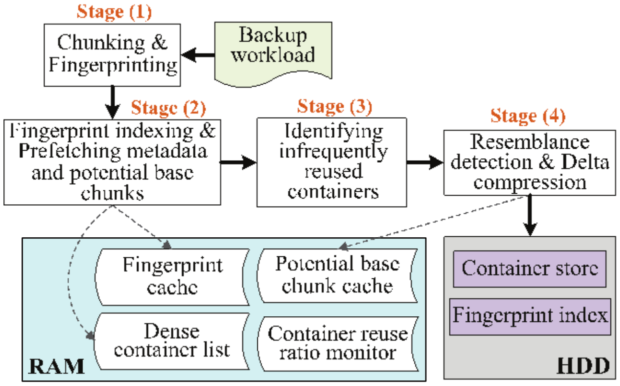
Actually, delta compression can be considered a process involving two steps. Specifically, the first step is to generate a delta for the target chunk, which leads to two versions of the target chunk: an encoded delta and an un-encoded one. The second step is to remove the un-encoded target chunk. If the target of delta compression is changed to previously written chunks, rather than data chunks in the ongoing backup as in the traditional delta compression method, there will be two versions of a previously written chunk. If the un-encoded one can be removed during GC, delta compression benefits will be obtained from the data chunks in infrequently reused containers without affecting the efficiency of rewriting.

# 4 Design and Implementation

## 4.1 LoopDelta Overview

LoopDelta is built on a typical deduplication strategy that groups data chunks into containers and accesses containers to prefetch metadata during deduplication to accelerate duplicate detection. It aims to embed delta compression in inline deduplication for highly efficient data reduction. The key idea behind LoopDelta is the combined use of the following four key techniques:

Dual-locality-based Similarity Tracking. By exploiting both the logical and physical locality based on the observations in Section 3.1, dual-locality-based similarity tracking identifies the containers that hold potential similar chunks, as detailed in Section 4.2.



Stage (1)
Chunking & Fingerprinting
Backup workload
Stage (2)
Fingerprint indexing & Prefetching metadata and potential base chunks
Stage (3)
Identifying infrequently reused containers
Stage (4)
Resemblance detection & Delta compression
Fingerprint cache
Potential base chunk cache
Dense container list
Container reuse ratio monitor
RAM
Container store
Fingerprint index
HDD

Figure 5: An overview of LoopDelta. The dashed arrows point to key data structures residing in DRAM required for the corresponding LoopDelta stages.

Locality-aware Prefetching. For containers holding potential similar chunks declared by the dual-locality-based similarity tracking, when they are accessed during deduplication to prefetch metadata, data chunks are also prefetched, i.e., piggybacked on the retrieval of the metadata to serve as potential base chunks, thereby avoiding extra I/Os for reading base chunks during backup, as detailed in Section 4.3.

Cache-aware Filter. LoopDelta identifies base-fragmented chunks with the assistance of recently prefetched metadata during deduplication and rewrites them to prevent extra I/Os for base chunks during restore, as detailed in Section 4.4.

Inversed Delta Compression. For data chunks whose similar chunks are prefetched from infrequently reused containers, LoopDelta delta-encodes the prefetched similar chunks relative to these data chunks while deferring the removal of the data of these delta encoded chunks to GC, as detailed in Section 4.5.

The overall workflow of the LoopDelta is illustrated in Figure 5, which includes four key stages. In stage (1), the backup stream is chunked and fingerprinted. Then, duplicate chunks are identified by indexing fingerprints in stage (2). In stage (2), potential base chunks and their sketches are loaded into the potential base chunk cache.

In stage (3), a rewriting approach is adopted to identify infrequently reused containers and fragmented chunks. This stage can be skipped to disable rewriting. Note that base-fragmented chunks are also identified in stages (2) and (3), as detailed in Section 4.4. In stage (4), for all unique, fragmented, and base-fragmented chunks, LoopDelta detects their similar chunks from the potential base chunk cache and performs delta compression for them if their base chunks exist. Finally, unremoved unique chunks and deltas are appended to a container.

In LoopDelta, a container consists of a metadata section and a data section, the same as that in [17, 53]. Data chunks and deltas are stored in the data section, while their metadata such as fingerprints, chunk length, and positions in the container are stored in the metadata section. We use a bitmap to specify which ones in the container are stored as deltas. For deltas, the metadata of their base chunks is also stored in the metadata section. For data chunks, their sketches are stored along with data chunks in the data section.

## 4.2 Dual-locality-based Similarity Tracking

The dual-locality-based similarity tracking is designed to identify similar chunks for delta-compressing the next backup. To capture similar chunks with logical locality, it tracks data chunks and base chunks of delta-compressed chunks of the ongoing backup. Meanwhile, to capture similar chunks with physical locality, it tracks data chunks stored in the same containers as the aforementioned data chunks.

Specifically, we define the reuse ratio of a container for a backup as the fraction of data chunks in this container referenced by this backup, $\frac{\text{the recorded chunk size}}{\text{the container size}}$, and use a container reuse ratio monitor to keep track of the reuse ratio of containers referenced by data chunks and base chunks of delta-compressed chunks of the on-going backup. When the backup completes, all containers holding potential similar chunks are recorded in the container reuse ratio monitor and are stored in a dense container list. Containers in this list will be prefetched to provide potential similar chunks for resemblance detection in the next backup.

Our approach for prefetching base chunks requires extra transfer time, thereby decreasing backup throughput, as detailed in the next subsection. To reduce this transfer time, only containers holding a large number of potential similar chunks can be recorded in the dense container list. Since containers with a larger reuse ratio are likely to contain more potential similar chunks for the next backup, we define a dense container threshold and only dense containers whose reuse ratios are greater than this threshold are included in the dense container list.

## 4.3 Locality-aware Prefetching

Locality-aware prefetching is designed to prefetch potential base chunks by piggybacking on routine operations for prefetching metadata during deduplication. LoopDelta adopts the duplicate detection strategy proposed by Zhu et al. [53], which employs an on-disk fingerprint index combined with an in-memory fingerprint cache and a Bloom filter for duplicate detection. Specifically, for each data chunk presented for storage, its fingerprint is compared against a fingerprint cache, and on a miss, a Bloom filter is checked to determine whether the data chunk is likely to exist in the system. If true, the on-disk fingerprint index is checked, and the metadata in the corresponding container is prefetched into the fingerprint cache. The fingerprints of the subsequent data chunks are like-ly to be matched in the fingerprint cache due to redundancy locality.

To prefetch potential base chunks by piggybacking on read operations for prefetching metadata, the dense container list of the last backup is loaded into memory to build a lookup table at the beginning of a backup. For each container to be accessed during deduplication, we check whether it exists in the dense container list generated by the last backup. If true, the whole container, including metadata and data chunks, is prefetched for both deduplication and delta compression; otherwise, only metadata are prefetched for deduplication. If the whole container is prefetched, all data chunks in the container as well as their sketches are inserted into the potential base chunk cache for resemblance detection. In LoopDelta, only non-delta-compressed chunks can serve as base chunks, as suggested by [37]. Thus, deltas are not loaded into the potential base chunk cache. When eviction occurs, based on the Least Recently Used (LRU) policy, data chunks and sketches from a container are evicted from the potential base chunk cache as a group.

Though locality-aware prefetching eliminates the seek and rotational delays of I/Os for reading base chunks, it increases the transfer time for prefetching data chunks in dense containers. The proposed dual-locality-based similarity tracking reduces this overhead by defining a dense container threshold, as detailed in the last subsection. Prefetching potential similar chunks according to a dense container list may be inefficient as containers in the list might have been reclaimed by GC, for which how to update the dense container list judiciously to minimize the problem will be discussed in Section 4.6. The previous technique for caching metadata on an SSD [1] is orthogonal to LoopDelta and could be used in LoopDelta to increase backup throughput.

## 4.4 Cache-aware Filter

Cache-aware filter is designed to identify base-fragmented chunks, which will be rewritten to achieve better restore performance. As introduced in Section 3.2.2, base-fragmented chunks can be identified with the assistance of the prefetched metadata during deduplication. Since the metadata prefetched by the routine operations during deduplication are loaded into the fingerprint cache, if the base chunk of a referenced previously written delta can be directly found in the restore cache during restore and thus does not require extra I/O, its fingerprint would also be directly found in the fingerprint cache in the fingerprint indexing stage.

When rewriting is applied, even if a base chunk's fingerprint can be directly found in the fingerprint cache, this base chunk can still trigger I/Os during restore. This is because a container whose metadata are prefetched into the fingerprint cache might be identified as an infrequently reused container by the rewriting approach in the next stage, i.e., stage (3) in Figure 5, so the data chunks it containers cannot serve as base chunks. Accordingly, the cache-aware filter adopts a two-step approach to identify base-fragmented chunks. First, it identifies the base-fragmented chunks referring to deltas whose base chunks do not exist in the fingerprint cache during fingerprint indexing. Then, it identifies the base-fragmented chunks referencing to deltas whose base chunks exist in the infrequently reused containers.

Specifically, in the fingerprint indexing stage, if a data chunk (say, CK) refers to a previously-written delta (say, Delta_old), the fingerprint of the delta's base chunk is fetched from the fingerprint cache and compared against the fingerprint cache. If the fingerprint does not exist in the fingerprint cache, CK is identified as a base-fragmented chunk; otherwise, the detected container ID (say, CID_base) of the base chunk is associated with CK. In the stage of identifying infrequently reused containers, if CK is identified as a fragmented chunk by the rewriting approach, there is no need to identify whether it is a base-fragmented chunk because it will be rewritten; otherwise, if CK is not a fragmented chunk, we further check whether the container whose ID is CID_base is selected to avoid being deduplicated against by the rewriting approach. If true, CK is identified as a base-fragmented chunk; otherwise, CK is identified as a duplicate chunk.

## 4.5 Post-deduplication Delta Compression

Inversed Delta Compression. The data chunks in the infrequently reused containers cannot serve as base chunks for delta compression; otherwise, the efficiency of the rewriting technique would be reduced. Inversed delta compression is designed to exploit the benefits of delta compression prohibited by rewriting. For a new chunk (say, N) that has a similar chunk (say, S) detected from the potential base chunk cache, the traditional direct delta compression approach delta-encodes N relative to S and generates a delta (say, Delta$_{n;s}$). Then, Delta$_{n;s}$ is stored instead of N to achieve data reduction.

On the contrary, inversed delta compression delta-encodes S relative to N and generates a delta (say, Delta$_{s;n}$), and then stores Delta$_{s;n}$ along with N. Since inversed delta compression generates an additional encoded S, the un-encoded S in the infrequently reused container will be removed during the next GC. It should be noted that delta-decoding a delta generated by inversed delta compression usually does not require extra I/Os for reading the base chunk because the delta (e.g., Delta$_{s;n}$) is stored together with the base chunk (e.g., N) in the same container, except that GC may occasionally disperse them to different containers.

Inversed delta compression increases the size of the data to be stored because it needs to store additional deltas, thereby increasing the I/O overheads for writing data relative to direct delta compression. Since LoopDelta is I/O-intensive, it only performs inversed delta compression for data chunks whose base chunks are prefetched from infrequently reused containers. Besides, inversed delta compression also causes more duplicate chunks to be removed during GC, which will be discussed in Section 4.6.

Table 1: 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 |
| CHM | 284GB | 100 versions of source codes of Chromium project from v84.0.4110 to v86.0.4215. Each version is combined into a tar file. | |
| SYN | 335GB | 180 versions of synthetic datasets generated by simulating file create /delete/modify operations. | Multi-version inheritance |

Delta Compression Workflow. For each unique, fragmented, and base-fragmented chunk, LoopDelta detects its base chunk from the potential base chunk cache and performs either direct or inversed delta compression according to whether rewriting is disabled or not. If it is disabled, LoopDelta only performs direct delta compression; otherwise, data chunks in the potential base chunk cache are divided into two categories: the ones prefetched from frequently reused containers and those prefetched from infrequently reused containers. When detecting base chunks, LoopDelta prefers the data chunks prefetched from frequently reused containers. Only if no such data chunk is detected, can data chunks prefetched from infrequently reused containers, if any, be selected as base chunks.

## 4.6 Garbage Collection

In the GC process, for a data chunk with multiple physical instances, the system marks the most recently written one as the live chunk. The delta-encoded chunks for inversed delta compression are removed in this process. GC may reduce the efficiency of locality-aware prefetching because containers in the dense container list generated by the proposed dual-locality-based similarity tracking in the last backup might have been reclaimed. To solve the problem, we update the dense container list to track potential base chunks after GC. Note that each backup stream has only one list to be updated. Thus, compared with GC, overheads for updating the dense container lists are negligible.

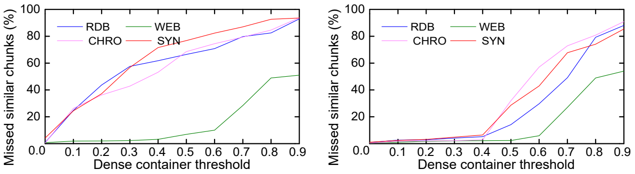
Moreover, inversed delta compression causes more duplicate chunks to be removed during GC, and the extra overhead introduced by inversed delta compression are negligible. This is because GC is time-consuming as it involves a large number of I/Os and the deduplication phase is not the bottleneck [10,14,17,40].

# 5 Performance Evaluation

## 5.1 Evaluation Setup

Experimental Platform. We perform our evaluation experiments on a workstation running Ubuntu 18.04 with an Intel Xeon(R) Silver 4215R CPU @ 3.20GHz, 32GB memory, Samsung 860 PRO SSDs, and Seagate 7200RPM SATA III HDDs.

System Configurations. For all approaches under evaluation, deduplication is configured to use the Rabin-based chunking algorithm [33,34] with the minimum, average, and maximum



(a) LoopDelta without Capping

(b) LoopDelta with Capping

Figure 6: Percentage of missed similar chunks as the dense container threshold varies from 0 through 0.9 on the four datasets. The Capping's capping level in this test is 15, namely, a 20MB segment (a sequence of consecutive data chunks) refers to at most 15 containers [22].

chunk sizes of 2KB, 8KB, and 64KB respectively for chunking and the SHA1 hash function for fingerprinting. The fingerprint cache has 256 slots to hold prefetched metadata. During restore, the restore cache is configured as a 512-container-sized (2GB) LRU cache.

For delta compression, we use Odess [54] for finding similar chunks and Xdelta [25] for delta encoding. For data chunks neither deduplicated nor delta-compressed, we compress them with a local compressor called ZSTD [13] before writing them into a container, the same as in [32]. The the container size is set to 4MB. To simulate backup and restore scenarios, an HDD is used as the backup space to store ingested data, and an SSD is used as the user space to store the original datasets.

Performance Metrics. We use three metrics to evaluate the performance of LoopDelta. The compression ratio is used to measure the total data reduction achieved by any compression technologies, including deduplication, delta compression, and local compression. It is calculated as $\frac{\text{original\_bytes}}{\text{post\_compression\_bytes}}$, so a compression ratio of greater than 1 means data reduction.

The speed factor (MB/container-read) is defined as the mean data size restored per container read [7, 8, 22], which is used to measure the restore performance. A larger speed factors indicates better restore performance. The backup throughput is measured by the throughput at which the input data are deduplicated, delta compressed, and written to the disk. We run each experiment five times to obtain a stable and average value of the backup throughput. Additionally, the shown speed factor is the average of the last 20 backups and the shown backup throughput is the average of the last 10 backups.

Evaluated Datasets. Four datasets, shown in Table 1 with their key characteristics, are used for performance evaluation. These datasets represent various typical workloads, including database snapshots, website snapshots, an open-source code project, and a synthetic dataset.

## 5.2 A Performance Study of LoopDelta

### 5.2.1 Dense Container Threshold

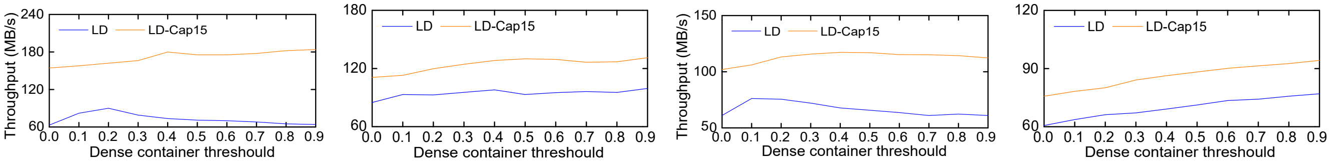
The dense container threshold can affect the number of detected similar chunks because it prevents containers whose reused ratios are smaller than it from being prefetched to supply potential similar chunks. Figure 6(a) suggests that, without rewriting, the percentage of similar chunks missed by LoopDelta increases quickly with the dense container threshold. One exception is the WEB dataset, where the percentage of missed similar chunks is low when the dense container threshold is small than 0.6. This is because this dataset contains substantial self-referenced duplicate chunks and most of the containers' reuse ratios are greater than 0.5.

Figure 6(b) suggests that Capping [22], a state-of-the-art rewriting approach introduced in Section 2.1, significantly decreases the percentage of missed similar chunks, especially when the dense container threshold is smaller than 0.4. By in-creasing the sequential layout of the current backup, rewriting improves both the logical and physical locality for the current and subsequent backups. This is why the percentage is low when the dense container threshold is small than 0.4. As the threshold increases beyond this, some containers holding data chunks that are inherited from the last backup are prevented from being prefetched for resemblance detection and thus the percentage of missed similar chunks grows quickly. When the dense container threshold is 0.3, LoopDelta only misses 1%-5% of similar chunks on the four datasets.

The dense container threshold can also affect backup throughput because it is related to two categories of I/O overheads: (1) transfer time for prefetching dense containers, which decreases as the threshold increases because fewer containers will be prefetched, and (2) container-writeback time saved by delta compression, which decreases as the threshold increases because fewer chunks will be delta compressed.

Figure 7 suggests that, except for the SYN dataset, the backup throughput hits a maximum and then either decreases or flattens out. This is because when the dense container threshold is very low, almost all containers holding potential similar chunks will be prefetched, leading to significant extra transfer time that exceeds the amount of container-writeback time saved by delta compression and resulting in very low throughput. This trend continues with the increase in dense container threshold until the decrease in extra transfer time for prefetching is offset by the decrease in the container-writeback time saved by delta compression. Beyond this point, the extra transfer time for prefetching becomes either greater than or equal to the decrease in container-writeback time saved by delta compression, causing the throughput to decrease or remain unchanged.

The SYN dataset contains only a few similar chunks that can be delta-compressed, and thus the container-writeback time saved by delta compression has a limited impact on backup throughput. Consequently, the backup throughput increases with the dense container threshold. Considering the two metrics (backup throughput and compression ratio) as a whole, in what follows, when rewriting (Capping) is applied, the dense container threshold is set to 0.3 as suggested by Figure 6(b) and discussed earlier.



(a) RDB (b) WEB (c) CHRO (d) SYN

Figure 7: Backup throughput as the dense container threshold varies from 0 through 0.9 on the four datasets. LD refers to LoopDelta without rewriting and LD-Cap15 refers to LoopDelta with Capping of a capping level of 15.

Table 2: Compression ratio and speed factor for LoopDelta (LD), with and without the cache-aware filter (CAF), and with and without rewriting (Capping), for the four datasets.

| Dataset | Approach | Compression ratio w/o GC | Speed factor |
|---|---|---|---|
| RDB | LD w/o CAF | 142.6 | 2.73 |
| | LD | 139.2 (-2.4%) | 2.81 (+2.9%) |
| | LD-Cap15 w/o CAF | 68.4 | 3.1 |
| | LD-Cap15 | 60.9 (-11%) | 4.67 (+50.6%) |
| WEB | LD w/o CAF | 112.2 | 2.81 |
| | LD | 105.7 (-5.8%) | 2.96 (+5.3%) |
| | LD-Cap15 w/o CAF | 51.2 | 6.84 |
| | LD-Cap15 | 46.9 (-8.4%) | 7.64 (+11.7%) |
| CHRO | LD w/o CAF | 70.7 | 2.66 |
| | LD | 70.3 (-0.6%) | 2.73 (2.6%) |
| | LD-Cap15 w/o CAF | 20.8 | 6.09 |
| | LD-Cap15 | 20.4 (-1.9%) | 8.12 (+33.3%) |
| SYN | LD w/o CAF | 33.9 | 0.84 |
| | LD | 33.8 (-0.3%) | 0.85 (+1.2%) |
| | LD-Cap15 w/o CAF | 17.1 | 1.38 |
| | LD-Cap15 | 15.4 (-9.9%) | 2.04 (+47.8%) |

### 5.2.2 Cache-aware Filter (CAF)

This subsection investigates the efficiency of the cache-aware filter (CAF). Since base-fragmented chunks identified by CAF will be rewritten, and rewritten data will be removed during GC. We do not run GC to show the trade-off between the decrease of compression ratio and the increase of speed factor caused by CAF.

Table 2 suggests that CAF slightly increases the speed factor (1.2%-5.3%, average of 3%) when rewriting is not applied, and it significantly increases the speed factor when rewriting is applied (by up to 50.6%, average of 35.9%), at the expense of a modest decrease (0.3%-11%, average of 4.8%) in compression ratio. CAF reduces the compression ratio because base-fragmented chunks are rewritten and not removed from storage as GC is not run. The decrease in compression ratio caused by CAF on the WEB and CHRO datasets is relatively small compared to the other two datasets. This is because these two datasets are tar-type files containing substantial small files, so most of their rewritten chunks can be delta-compressed.

A larger fingerprint cache can help to identify more base-fragmented chunks at the cost of increased computational overheads. Since the bottleneck of LoopDelta lies in I/O overheads, the computational overhead of CAF has almost no impact on backup throughput. We found that, most base-fragmented chunks can be identified by the first few slots at the front of the fingerprint cache due to locality, especially when rewriting is applied. To reduce the computational overhead, we suggest using the first 20 slots of the fingerprint cache to identify base-fragmented chunks when rewriting is applied and using the first 64 slots when rewriting is not applied. This strategy does not compromise restore performance because base-fragmented chunks that are not identified will still be recognized due to their container IDs not being found in the fingerprint cache.

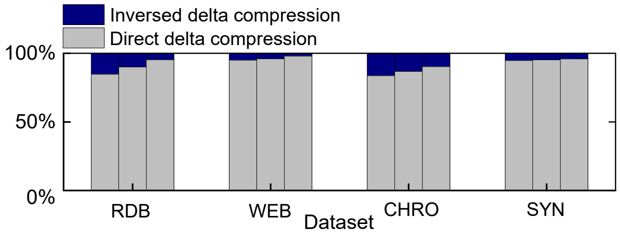
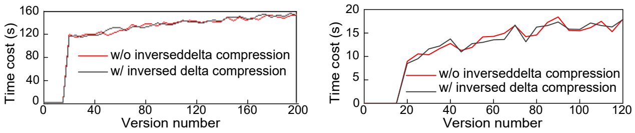


Figure 8: Proportion of compression ratio achieved by direct and inversed delta compression with different capping level on the four datasets. The three bars on each dataset from left to right represent the three capping levels of 10, 15, and 20, respectively.



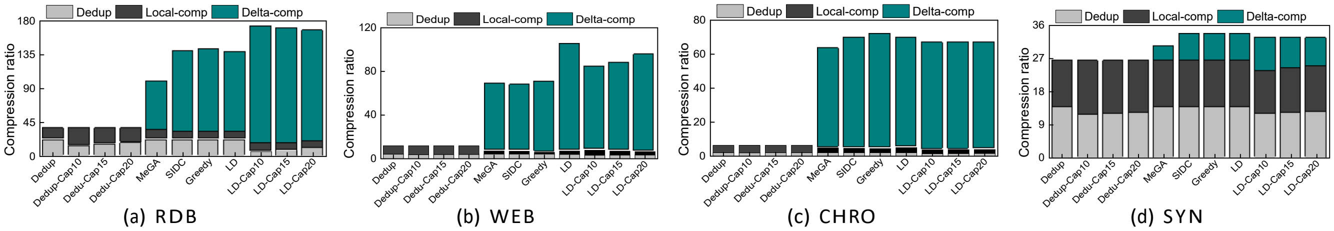
(a) RDB (b) WEB

Figure 9: Time cost of GC for LD-Cap10 with and without inversed delta compression on the RDB and WEB datasets.

### 5.2.3 Inversed Delta Compression

This subsection investigates the efficiency of inversed delta compression. Figure 8 suggests that the compression gains achieved by inversed delta compression account for 2.2%-16.4% of the combined compression gains by direct and in-versed delta compression. For example, for LD-Cap10, the compression ratio achieved by inversed delta compression accounts for 15.3%, 5%, 16.4%, and 5.3% respectively of that achieved jointly by direct and inversed delta compression on the four datasets.

Inversed delta compression causes more data to be stored because it needs to store the deltas of data chunks being delta-encoded. For LD-Cap10, extra stored data caused by inversed delta compression account for 2.1%, 0.8%, 2.7%, and 0.1% of



(a) RDB (b) WEB (c) CHRO (d) SYN

Figure 10: Comparison of compression ratio achieved by the eleven approaches on the four datasets.

the total stored data on the four datasets respectively, which are marginal. This is because the size of a delta is often much smaller than that of a data chunk compressed with the local compressor, e.g., the former is 1/26-1/10 of the latter in size in our test.

Inversed delta compression also causes more data chunks to be removed during GC. Figure 9 compares the time cost of LD-Cap10 with and without inversed delta compression on the RDB and WEB datasets. To accumulate more deltas generated by inversed delta compression, we run GC after every 5 backups from the $20_{th}$ backup. The results in Figure 9 suggest that inversed delta compression has a negligible impact on the time cost of GC because the bottleneck of GC lies in marking and moving forward the live chunks, which requires a large number of I/Os.

## 5.3 Comprehensive Evaluation of LoopDelta

In this section, we comprehensively evaluate the performance of LoopDelta in terms of three key metrics: compression ratio, speed factor, and backup throughput. Five data reduction approaches are also tested: Dedup, Dedup-Cap, MeGA, SIDC, and Greedy. Specifically, Dedup is a deduplication approach proposed by Zhu et al. [53] without rewriting, and Dedup-Cap refers to Dedup with Capping. MeGA, SIDC, and Greedy have been discussed in Section 3.1. Dedup-Cap# and LD-Cap# represent Dedup-Cap and LD-Cap with a different capping level of #.

MeGA's restore performance and backup throughput are not evaluated because it requires additional offline and service-disruptive operations. In contrast, LoopDelta focuses on adding delta compression to inline deduplication systems in a non-service-disruptive manner. In evaluations of this section, we use a 20-container-sized base-chunk cache for LD-Cap#, and a 150-container-sized base-chunk cache for MeGA, SIDC, Greedy, and LD. With rewriting (Capping), a 20-container-sized cache can capture almost all similar chunks (not shown due to space limit).

Compression Ratio. In this test, we run GC after each backup from the $20_{th}$ backup. Figure 10 suggests that LD achieves a compression ratio comparable to SIDC and Greedy and higher than MeGA on the RDB, CHRO, and SYN datasets. This means that LD can detect most potential similar chunks by exploiting both logical and physical locality. On the WEB dataset, LD achieves the highest compression ratio because it can detect more highly similar chunks than the other approaches on the datasets containing self-referenced similar chunks. Specifically, LD outperforms MeGA, SIDC, and Greedy by 1.55, 1.56, and 1.5, respectively, on the WEB dataset. Furthermore, LD achieves significantly higher compression ratios than Dedup, with improvements of 3.81 (RDB), 8.9 (WEB), 10.97 (CHRO), and 1.27 (SYN).

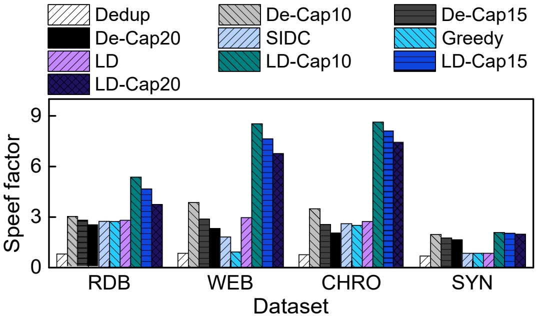


Figure 11: Comparison of speed factor achieved by the ten approaches on the four datasets.

Additionally, LD-Cap15 achieves 4.68 (RDB), 7.42 (WEB), 10.51 (CHRO), and 1.24(SYN) higher compression ratio than Dedup-Cap15, respectively. Rewriting decreases the compression ratio of LD because a small number of similar chunks are missed, as analyzed in Section 5.2.1. One exception is the RDB dataset, where rewriting increases the compression ratio. This is because rewriting causes the rewritten chunks to be detected as base chunks. Compared to the data chunks written much earlier, rewritten chunks are more similar to the data chunks in the current backup. In our tests, the average DCE of LD and LD-Cap15 are 0.9398 and 0.9591, respectively. Note that while the total amount of redundancy eliminated by LD-Cap15 may not be significantly more than that eliminated by LD, compressing even a small amount of additional data can lead to a significant increase in compression ratio when the compression ratio is very high.

Note that LD-Cap# may achieve smaller deduplication gains and more delta compression gains than the other approaches. This is because LD-Cap# rewrite fragmented chunks and perform delta compression for them, which reduces deduplication gains but increases delta compression gains.

Speed Factor. Figure 11 suggests that LD achieves the highest speed factor among all approaches without rewriting and LD-Cap# also achieve a higher speed factor than Dedup-Cap#. Specifically, LD achieves 3.48 (RDB), 3.47 (WEB), 3.57 (CHRO), and 1.24 (SYN) higher speed factor than

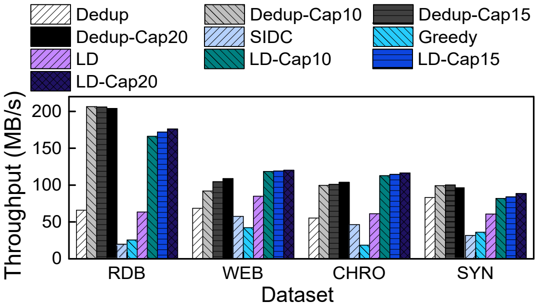


Figure 12: Comparison of backup throughput achieved by the ten approaches on the four datasets.

Dedup respectively, and LD-Cap15 achieves 1.68 (RDB), 2.65 (WEB), 3.17 (CHRO), and 1.2 (SYN) higher speed factor than Dedup-Cap15 respectively.

Generally, approaches combining deduplication and delta compression, i.e., SIDC, Greedy, and LD, achieve a higher speed factor than Dedup, which only performs deduplication. This is because delta compression has the potential to increase restore performance, which mainly depends on the number of read-in containers during restore. Delta compression decreases the number of written containers during backup and the number of read-in containers during restore. However, if base chunks require extra I/Os during restore, the improvement in speed factor resulting from delta compression will decrease. In LD, base chunks do not require extra I/Os during restore, and this is ensured by CAF.

Greedy detects similar chunks without considering the positions of base chunks. Thus, base chunks in it may require substantial I/Os during restore and this will lead to a lower speed factor, e.g., on the WEB dataset. SIDC detects similar chunks stored along with duplicate chunks, and thus, the base chunks of the delta-compressed chunks in it do not require extra I/Os during restore. However, SIDC fails to identify base-fragmented chunks. This is why LD achieves a slightly higher speed factor than SIDC on the RDB, CHRO, and SYN datasets. LD also achieves a higher speed factor than SIDC on the WEB dataset because it achieves higher compression ratio than SIDC, which means fewer read-in containers during restore.

Backup Throughput. Figure 12 suggests that LD and LD-Cap# achieve lower backup throughput than Dedup and Dedup-Cap# respectively on the RDB and SYN datasets and higher backup throughput than them on the WEB and CHRO datasets. Specifically, LD achieves 3.6% and 27.1% lower backup throughput than Dedup on the RDB and SYN datasets, and 24% and 10.5% higher backup throughput than Dedup on the WEB and CHRO datasets. Meanwhile, LD-Cap15 achieves 16.5% and 16.3% lower backup throughput than Dedup-Cap15 on the RDB and SYN datasets, and 13.9% and 13.5% higher backup throughput than Dedup-Cap15 on the WEB and CHRO datasets. Additionally, LD (LD-Cap15) achieves 1.33.7 (2.39.9) higher backup through-put than SIDC and Greedy on the four datasets. This is because LoopDelta eliminates the seek and rotational delays of I/Os for reading base chunks.

Compared with deduplication-based backup systems, post-deduplication delta compression adds additional computational overheads, but the backup throughput is mainly decided by I/O overheads. In LoopDelta, multiple tasks in its workflow involve I/Os, including (1) looking up the fingerprint index, (2) prefetching metadata, (3) prefetching potential base chunks, (4) updating the fingerprint index, and (5) writing back containers. For datasets with low redundancy, such as the WEB and CHRO datasets, there are more unique chunks that lead to more I/Os in tasks (4) and (5), making them the performance bottleneck. Delta compression increases the backup throughput on such datasets because it alleviates the performance bottleneck by reducing I/Os in tasks (5). For datasets with high redundancy, such as the RDB and SYN datasets, there are more duplicate chunks that result in more I/Os in tasks (1), (2), and (3), making them the performance bottleneck. Delta compression decreases backup throughput because it aggravates the performance bottleneck by increasing I/Os in task (3).

To sum up, LoopDelta achieves a comparable or higher compression ratio, higher restore performance, and higher backup throughput than the other post-deduplication delta compression approaches. It also increases the compression ratio by 1.2410.97 times on top of deduplication, without notably affecting backup throughput, and improves the restore performance by 1.23.57 times.

## 6 Conclusion

In this paper, we present LoopDelta to embed delta compression in inline deduplication. The key idea of LoopDelta is the combined use of four key techniques, i.e., dual-locality-based similarity tracking to detect similar chunks, locality-aware base prefetching to avoid extra I/Os for reading base chunks on the write path, cache-aware filter to avoid extra I/Os for reading base chunks on the read path, and inversed delta compression to perform delta compression for similar chunks prefetched from infrequently reused containers. The experimental results indicate that LoopDelta significantly increases compression ratio and improves restore speed over deduplication, without notably affecting backup throughput.

## 7 Acknowledgment

We are grateful to our shepherd and the anonymous reviewers for their insightful comments and feedback on this work. This research was partly supported by the National Natural Science Foundation of China No. 61821003, No. 61832007, No. 62262042, No. 62271239, No. 62172160, and No. 62062034; the US NSF CNS-2008835; the Jiangxi Provincial Natural Science Foundation No. 20224BAB202017, No. 20212ACB212002, and No. 20224ABC03A01; the Open Project Program of Wuhan National Laboratory for Optoelectronics No. 2021WNLOKF012.

## References

[1] Yamini Allu, Fred Douglis, Mahesh Kamat, Ramya Prabhakar, Philip Shilane, and Rahul Ugale. Can't we all get along? redesigning protection storage for modern workloads. In the 2016 conference on USENIX Annual Technical Conference (ATC'18, pages 705–718, Boston, MA, USA, July 11 - 13 2018. USENIX Association.

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

[3] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In the 2015 conference on USENIX Annual Technical Conference (ATC'15), pages 151–164, Santa Clara, CA, July 08 - 10 2015. USENIX Association.

[4] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. The design of a similarity based deduplication system. In the 2th Annual International Systems and Storage Conference (SYSTOR'09), pages 1–14, Haifa, Israe, May 04 - 06 2009. ACM Association.

[5] Fabiano C Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In the 11th USENIX Conference on File and Storage Technologies (FAST'13), pages 81–94, San Jose, CA, February 12 - 15 2013. USENIX Association.

[6] Andrei Z Broder. Identifying and filtering near-duplicate documents. In Annual Symposium on Combinatorial Pattern Matching, pages 1–10. Springer, 2000.

[7] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David HC Du. 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), pages 129–142, Boston, MA, USA, February 25 - 28 2019. USENIX Association.

[8] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. 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), pages 309–324, Oakland, CA, USA, February 12 - 15 2018. USENIX Association.

[9] Biplob Debnath, Sudipta Sengupta, and Jin Li. ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory. In the 2010 conference on USENIX Annual Technical Conference (ATC'10), pages 1–16, Boston, MA, USA, June 23 - 25 2010. USENIX Association.

[10] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano Botelho. The logic of physical garbage collection in deduplicating storage. In the 15th USENIX Conference on File and Storage Technologies (FAST'17), pages 29–44, Santa Clara, CA, USA, February 12 - 15 2017. USENIX Association.

[11] Fred Douglis and Arun Iyengar. Application-specific delta-encoding via resemblance detection. In the 2003 USENIX conference on USENIX Annual Technical Conference (ATC'03), pages 113–126, San Antonio, TX, USA, June 09 - 14 2003. USENIX Association.

[12] Kruus Erik, Ungureanu Cristian, and Dubnicki Cezary. Bimodal Content Defined Chunking for Backup Streams. In the 8th USENIX Conference on File and Storage Technologies (FAST'10), pages 1–14, San Jose, CA, US-A, February 23 - 26 2010. USENIX Association.

[13] Facebook. Zstandard. https://github.com/facebook/zstd, September 2022. zstd.

[14] Min Fu, Dan Feng, Yu Hua, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. 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), pages 181–192, Philadelphia, PA, USA, June 19 - 20 2014. USENIX Association.

[15] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. IEEE Transactions on Parallel and Distributed Systems, 27(3):855–868, 2015.

[16] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In the 13th USENIX Conference on File and Storage Technologies (FAST'15), pages 331–345, Santa Clara, CA, USA, February 16 - 19 2015. USENIX Association.

[17] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In the 2011 USENIX conference on USENIX Annual Technical Conference (ATC'11), pages 1–14, Portland, OR, USA, June 15 - 17 2011. USENIX Association.

[18] Diwaker Gupta, Sangmin Lee, Michael Vrable, and et al. Difference engine: Harnessing memory redundancy in virtual machines. In the 5th Symposium on Operating Systems Design and Implementation (OSDI'08), pages 309–322, San Diego, CA, USA, December 08 - 10 2008. USENIX Association.

[19] Navendu Jain, Michael Dahlin, and Renu Tewari. TAPER: Tiered Approach for Eliminating Redundancy in Replica Synchronization. In the 3th USENIX Conference on File and Storage Technologies (FAST'05), pages 281–294, San Francisco, CA, USA, March 13 - 16 2005. USENIX Association.

[20] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In the 5th Annual International Systems and Storage Conference (SYSTOR'12), pages 1–12, Haifa, Israe, June 04 - 06 2012. ACM Association.

[21] Purushottam Kulkarni, Fred Douglis, Jason D LaVoie, and John M Tracey. Redundancy elimination within large collections of files. In the 2004 USENIX Annual Technical Conference (ATC'04), pages 59–72, Boston, MA, USA, June 27 - July 01 2004. USENIX Association.

[22] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In the 11th USENIX Conference on File and Storage Technologies (FAST'13), pages 183–197, San Jose, CA, USA, February 12 - 15 2013. USENIX Association.

[23] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezise, and Peter Camble. Sparse indexing: Large scale, inline deduplication using sampling and locality. In the 7th USENIX Conference on File and Storage Technologies (FAST'09), volume 9, pages 111–123, San Jose, CA, February 24 - 27 2009. USENIX Association.

[24] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: Coarse-grained data reordering to improve compressibility. In the 12th USENIX Conference on File and Storage Technologies (FAST'14), pages 257–271, Santa Clara, CA, USA, February 17 - 20 2014. USENIX Association.

[25] Josh MacDonald. File system support for delta compression. PhD thesis, Masters thesis. Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[26] Dirk Meister and Andre Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10), pages 1–6, Incline Village, Nevada, USA, May 03 - 07 2010. IEEE Computer Society Press.

[27] Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block locality caching for data deduplication. In the 6th International Systems and Storage Conference (SYSTOR'13), pages 1–12, Haifa, Israel, June 30 - July 02 2013. ACM Association.

[28] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In the 9th USENIX Conference on File and Storage Technologies (FAST'11), pages 229–241, San Jose, CA, USA, February 15 - 17 2011. USENIX Association.

[29] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A Low-Bandwidth Network File System. In the ACM Symposium on Operating Systems Principles (SOSP'01), pages 1–14, Banff, Canada, October 21 - 24 2001. ACM Association.

[30] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. 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), pages 581–586, Banff, Canada, September 02 - 04 2011. IEEE Computer Society Press.

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

[32] Jisung Park, Jeonggyun Kim, Yeseong Kim, Sungjin Lee, and Onur Mutlu. 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), pages 247–264, Santa Clara, CA, USA, February 22 - 24 2022. USENIX Association.

[33] Sean Quinlan and Sean Dorward. Venti: a New Approach to Archival Storage. In the 1st USENIX Conference on File and Storage Technologies (FAST'02), pages 89–101, Monterey, CA, USA, January 28 - 30 2002. USENIX Association.

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

[35] B. Romański, Ł. Heldt, W. Kilian, K. Lichota, and C. Dubnicki. Anchor-driven subchunk deduplication. In

The 4th Annual International Systems and Storage Conference (SYSTOR'11), pages 1–13, Haifa, Israel, May 30 - June 01 2011. ACM Association.

[36] Shruti Sanadhya, Raghupathy Sivakumar, Kyu-Han Kim, Paul Congdon, Sriram Lakshmanan, and Jatinder Pal Singh. Asymmetric caching: Improved network deduplication for mobile devices. In the 18th annual international conference on Mobile computing and networking (MobiCom'12), pages 161–172, Istanbul, Turkey, August 22 - 26 2012. ACM Association.

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

[38] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta Compressed and Deduplicated Storage Using Stream-Informed Locality. In the 4th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'12), Boston, MA, USA, June 13 - 14 2012. USENIX Association.

[39] Yujuan Tan, Jian Wen, Zhichao Yan, Hong Jiang, Srisaan Witawas, Baiping Wang, and Hao Luo. Fgdefrag: A fine-grained defragmentation approach to improve restore performance. In the 33th Symposium on Mass Storage Systems and Technologies (MSST'17), Santa Clara, California, May 15 - 19 2017. IEEE Computer Society Press.

[40] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Cumulus: Filesystem backup to the cloud. In the 7th USENIX Conference on File and Storage Technologies (FAST'09), pages 225–238, Santa Clara, CA, USA, February 24 - 27 2009. USENIX Association.

[41] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In the 10th USENIX Conference on File and Storage Technologies (FAST'12), pages 1–14, San Jose, CA, February 14 - 17 2012. USENIX Association.

[42] Chunzhi Wang, Yanlin Fu, Junyi Yan, Xinyun Wu, Yucheng Zhang, Huiling Xia, and Ye Yuan. A cost-efficient resemblance detection scheme for post-deduplication delta compression in backup systems. Concurrency and Computation: Practice and Experience, 2021.

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

[44] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. 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), pages 285–298, Portland, OR, June 15 - 17 2011. USENIX Association.

[45] Wen Xia, Chunguang Li, Hong Jiang, Dan Feng, Yu Hua, Leihua Qin, and Yucheng Zhang. Edelta: A word-enlarging based fast delta compression approach. In the 7th USENIX conference on Hot Topics in Storage and File Systems (HotStorage'15), Santa Clara, CA, July 06 - 07 2015. USENIX Association.

[46] Wen Xia, Yukun Zhou, Hong Jiang, Dan Feng, Yu Hua, Yuchong Hu, Qing Liu, and Yucheng Zhang. FastCDC: A fast and efficient content-defined chunking approach for data deduplication. In the 2016 conference on USENIX Annual Technical Conference (ATC'16), pages 101–114, Denver, CO, June 15 - 17 2016. USENIX Association.

[47] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online deduplication for databases. In ACM International Conference on Management of Data (SIGMOD'17), pages 1355–1368, Chicago, IL, USA, 2017. ACM Association.

[48] Lawrence L You, Kristal T Pollack, and Darrell DE Long. Deep store: An archival storage system architecture. In the 21st International Conference on Data Engineering (ICDE'05), pages 804–815, Tokyo, Japan, April 5 - 8 2005. IEEE.

[49] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. 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), pages 1337–1345, Hong Kong, China, April 26th - May 1st, 2015. IEEE.

[50] Yucheng Zhang, Hong Jiang, Mengtian Shi, Chunzhi Wang, Nan Jiang, and Xinyun Wu. A high-performance post-deduplication delta compression scheme for packed datasets. In IEEE 39th International Conference on Computer Design (ICCD'21), pages 464–471. IEEE, October 24 - 27 2021.

[51] Yucheng Zhang, Wen Xia, Dan Feng, Hong Jiang, Yu Hua, and Qiang Wang. 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),

pages 121–128, Boston, MA, USA, February 25 - 28 2019. USENIX Association.

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

[53] Benjamin Zhu, Kai Li, and Patterson Hugo. Avoiding the disk bottleneck in the data domain deduplication file system. In the 6th USENIX Conference on File and Storage Technologies (FAST'08), pages 269–282, San Jose, CA, USA, February 26 - 29 2008. USENIX Association.

[54] Xiangyu Zou, Cai Deng, Wen Xia, Philip Shilane, Haoliang Tan, Haijun Zhang, and Xuan Wang. Odess: Speeding up resemblance detection for redundancy elimination by fast content-defined sampling. In the 37th International Conference on Data Engineering (ICDE'21), pages 480–491. IEEE, April 19 - 22 2021.

[55] Xiangyu Zou, Wen Xia, Philip Shilane, Haijun Zhang, and Xuan Wang. Building a high-performance fine-grained deduplication framework for backup storage with high deduplication ratio. In the 2022 USENIX Annual Technical Conference (ATC'22), pages 19–36, Carlsbad, CA, USA, July 11 - 13 2022. USENIX Association.

[56] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The dilemma between deduplication and locality: Can both be achieved? In the 19th USENIX Conference on File and Storage Technolo-gies (FAST'21), pages 171–185. USENIX Association, February 23 - 25 2021.