



Latest updates: <https://dl.acm.org/doi/10.1145/3689031.3717493>

RESEARCH-ARTICLE

## Garbage Collection Does Not Only Collect Garbage: Piggybacking-Style Defragmentation for Deduplicated Backup Storage

**DINGBANG LIU**, Harbin Institute of Technology Shenzhen, Shenzhen, Guangdong, China

**XIANGYU ZOU**, Harbin Institute of Technology Shenzhen, Shenzhen, Guangdong, China

**TAO LU**

**PHILIP SHILANE**, Dell Inc., Round Rock, TX, United States

**WEN XIA**, Harbin Institute of Technology Shenzhen, Shenzhen, Guangdong, China

**WENXUAN HUANG**, Harbin Institute of Technology Shenzhen, Shenzhen, Guangdong, China

[View all](#)

**Open Access Support** provided by:

**Harbin Institute of Technology Shenzhen**

**Dell Inc.**



PDF Download  
3689031.3717493.pdf  
26 February 2026  
Total Citations: 1  
Total Downloads: 1417



Published: 30 March 2025

[Citation in BibTeX format](#)

EuroSys '25: Twentieth European Conference on Computer Systems  
March 30 - April 3, 2025  
Rotterdam, Netherlands

Conference Sponsors:  
**SIGOPS**



# Garbage Collection Does Not Only Collect Garbage: Piggybacking-Style Defragmentation for Deduplicated Backup Storage

Dingbang Liu

Harbin Institute of Technology,  
Shenzhen, China  
nickluisliu@outlook.com

Philip Shilane

Dell Technologies  
Boston, USA  
Philip.Shilane@dell.com

Yanqi Pan

Harbin Institute of Technology,  
Shenzhen, China  
deadpooldeathmine@gmail.com

Xiangyu Zou\*

Harbin Institute of Technology,  
Shenzhen, China  
zouxiangyu@hit.edu.cn

Wen Xia

Harbin Institute of Technology,  
Shenzhen, China  
xiawen@hit.edu.cn

Tao Lu

DapuStor  
Shenzhen, China  
lvtao@dapustor.com

Wenxuan Huang

Harbin Institute of Technology,  
Shenzhen, China  
nniferry@gmail.com

Hao Huang

Harbin Institute of Technology,  
Shenzhen, China  
haohuang.cs@foxmail.com

## Abstract

Deduplication is widely used in backup storage and reduces storage overhead by allowing backups to share common data chunks. However, it naturally disrupts the sequential layout of backup images, leading to fragmentation, which slows down backup restoration. Existing solutions to this issue often come with trade-offs, either reducing deduplication effectiveness or introducing significant I/O overhead.

In this paper, we propose GCCDF, a novel approach that enhances the efficiency of deduplication-based backup storage. It reorders data as part of garbage collection to eliminate fragmentation, avoiding additional I/O costs. During the reordering, it effectively groups related data for better locality and aligns with the storage layout in backup storage. Evaluation results demonstrate that GCCDF significantly improves restoration speed, offsets data migration overhead, and preserves the deduplication ratio.

**CCS Concepts:** • Information systems → Deduplication; • Software and its engineering → Garbage collection;

\*Corresponding author.

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.  
*EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands*

© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-1196-1/25/03...\$15.00  
<https://doi.org/10.1145/3689031.3717493>

## Backup procedures.

### ACM Reference Format:

Dingbang Liu, Xiangyu Zou, Tao Lu, Philip Shilane, Wen Xia, Wenxuan Huang, Yanqi Pan, and Hao Huang. 2025. Garbage Collection Does Not Only Collect Garbage: Piggybacking-Style Defragmentation for Deduplicated Backup Storage. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands*. ACM, New York, NY, USA, 18 pages. <https://doi.org/10.1145/3689031.3717493>

## 1 Introduction

Backup storage, particularly Purpose-Built Backup Appliances (PBBA), plays a critical role in enterprise data protection, ensuring reliable backups and rapid recovery amid increasing cyber threats [15]. However, the explosive growth of backup data [7] has made storage costs a major concern [14, 16]. This makes it crucial to incorporate data reduction techniques into backup storage.

Deduplication [5, 9, 22, 24, 33, 35, 50] is a typical data reduction technique that eliminates redundant data chunks by replacing them with reference pointers, ensuring that only unique chunks are stored. It has become a cornerstone of backup storage, reducing storage footprint by over 90%.

However, deduplication introduces a critical challenge: it disrupts the logical organization of backup images. Since it replaces data chunks with random pointers, deduplication disorganizes what would otherwise be sequential storage. The resulting lack of locality is called deduplication-specific fragmentation. Figure 1 illustrates this, depicting a scattered storage layout caused by the interleaving of unique and referenced historical chunks. It severely impacts data restoration performance by inducing random I/O operations and I/O amplification [25]. As a result, deduplicated backups can

suffer up to an 80% reduction in restoration speed (§ 3.1), making restoration efficiency a pressing concern for enterprises [14, 16]. Given that downtime costs can reach \$250K per hour [14], optimizing restoration speed in deduplicated backup systems is imperative (§ 2.1).

Existing techniques aim to mitigate deduplication-specific fragmentation, but suffer from multiple drawbacks. One the one hand, *rewriting-based methods* [12, 25, 42] achieve only marginal restoration speed improvements, but suffer huge deduplication ratio loss. On the other hand, *reordering-based methods* [53, 54] incur substantial I/O overhead, typically copying data of 50%–80% of the total size of the dataset. It increases the I/O burden of the backup system, and degrades overall system performance.

Can we design a solution that overcomes the limitations of existing techniques and simultaneously optimizes restoration speed, deduplication ratio, and I/O overhead?

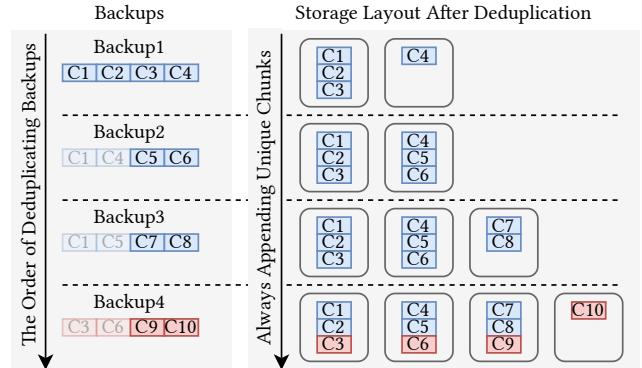
Our key insight is to integrate reordering-based defragmentation with garbage collection (GC), leveraging their inherent similarities. GC is an essential periodic process in deduplication systems. It identifies valid chunks, and reclaims storage space by *data migration*, which copies valid chunks to new locations, and eliminates obsolete ones. Similarly, reorder-based defragmentation relies on data migration to copy and reorganize correlated data together. Both involve data migration, presenting an opportunity to consolidate their operations. By embedding defragmentation within GC, we can hide most overhead contributed by reordering.

Accordingly, we propose GCCDF, a garbage-collection-collaborative defragmentation approach that enhances restoration speed in deduplicated backup storage. GCCDF leverages GC’s data migration to optimize data layout, significantly reducing fragmentation while preserving deduplication efficiency. GCCDF addresses two key challenges:

① *Conflicts of locality between backups*. In deduplicated storage, multiple backups share unique data chunks, leading to locality conflict: Consolidating these chunks by the sequence of one backup often disrupts the locality of another. GCCDF introduces a *locality-compatible chunk clustering* strategy. It evaluates the compatibility of locality among all backups and identifies a way to cluster chunks that minimizes fragmentation across all backups. This strategy effectively mitigates fragmentation globally while maintaining the deduplication ratio.

② *Granularity mismatch between chunk clustering and storage units*. Backup storage systems use fixed-sized containers as fundamental I/O units, but our defragmentation operates on variable-sized chunk clusters, leading to misalignment. GCCDF introduces a *container-adaptable cluster packing* strategy, which minimizes the impact thereof.

Evaluation shows that GCCDF outperforms state-of-the-art solutions. Specifically, it achieves a 2.1× speedup over SMR [42] in backup restoration while avoiding the deduplication ratio loss observed in SMR (up to 34.5%). In addition,



**Figure 1.** Deduplication-specific fragmentation occurs when backups (left) are deduplicated and arranged in storage layout (right): Backup4 (red-shaded) becomes scattered naturally. Duplicate chunks are shown in semi-transparent color.

GCCDF also achieves a 6.45× higher deduplication ratio than MFDedup [54] in typical scenarios.

The contributions of this paper are threefold:

- Identifying the opportunity to integrate GC and defragmentation operations to reduce their associated overhead.
- Exploring the compatibility of locality among deduplicated backups while preserving their deduplication ratio.
- Minimizing the impact of misalignment between data units in storage layout and defragmentation operations.

## 2 Background and Related Work

### 2.1 Backup Storage

Backup storage (i.e., Purpose-Built Backup Appliance, PBBA) [1, 2, 37, 38] is usually a standalone storage server that is optimized for reliable backup and efficient recovery. As cyber threats and data breaches continue to rise, enterprises rely on backup solutions to safeguard critical data.

**Container-based Layout.** In backup storage, data is usually split into chunks. These chunks are then stored in *containers*, which are data structures that hold chunks. A container is immutable, and has a large, fixed size (e.g., 4MB).

Containers are widely used on RAID-based storage in commercial backup products from Dell [51], HP [26], Symantec [17], Microsoft [8], etc. They offer several benefits: ① Writing in large units (containers, rather than chunks) maximizes hard drive throughput and aligns with RAID striping [25, 40, 49, 52]. ② Some deduplication techniques structure the index at the granularity of containers for the identification of duplicate chunks [49, 52].

**The Need for Fast Restoration in Backup Storage.** Restoration speed is considered critical on selecting a disaster recovery solution [16]. This is particularly important given that backup and recovery processes still face significant challenges [14, 15], primarily due to the substantial cost of downtime, which averages \$250,000 per hour [14]. Such economic loss has led approximately 21% of surveyed ransomware victims to opt for paying the ransom, even when backups are

available, to expedite recovery [16].

## 2.2 Deduplication in Backup Storage

To reduce redundancy, deduplication in backup storage performs the following steps [4, 11, 17, 53, 54]: ① Split a backup image into chunks, which can be fixed-sized or variable-sized, typically 8KB or larger. ② Remove chunks duplicated in both the same backup and previous deduplicated backups. ③ Compress and store unique chunks consecutively in containers. ④ Create a *recipe* for each deduplicated backup that records its component chunks. ⑤ Reclaim chunks that become unused (i.e., no longer referenced by any backup) due to deleted backups periodically through GC.

Deduplication saves storage space by allowing backup images to share common chunks. However, it also introduces certain drawbacks, as mentioned in the following subsection.

## 2.3 Fragmentation in Deduplicated Backup Storage

Although fragmentation is common for storage systems, it is quite different in deduplicated backup storage.

In backup storage, a backup image is usually archived in tar format [13], which is large-sized [10] and immutable. These images are periodically rotated in backup storage (i.e., older versions are replaced according to a predefined policy). Frequent updates, the cause of fragmentation in classical storage, are not commonly seen in backup storage.

Instead, the main cause of fragmentation is *chunk sharing across images*. Specifically, because duplicate chunks of backups are not stored, backups refer to existing replicas of them, which scatters logically sequential chunks. Fig. 1 illustrates four backup images consisting of multiple chunks, some of which are duplicates. Take Backup4 for example. C3 and C6 of this backup are duplicate, so it only refers to the existing replicas of C3 and C6. Eventually, the chunks of Backup4 are scattered across containers.

Fragmentation destroys the locality of backups by scattering chunks. Due to container-based I/O, restoring Backup4 requires reading all four associated containers from disk, even though only four chunks within them are actually needed. This causes read amplification during restoration and constrains restoration performance.

**Defragmentation for Deduplicated Storage.** Defragmentation in deduplicated storage fundamentally differs from classical storage defragmentation. While both aim to optimize data layout, the criteria for grouping data differ:

In *classical storage*, a block belongs to a single file, enabling straightforward physical adjacency optimization for sequentially-accessed files.

In *deduplicated storage*, a chunk may be shared across multiple backups, creating interdependencies. Optimizing chunk placement for one backup often introduces fragmentation to others, as discussed in § 3.3.

## Defragmentation Approaches for Deduplicated Backup Storage.

*Rewriting* and *reordering* are two kinds of approaches to address fragmentation in deduplicated backup storage.

*Rewriting* [12, 25, 42] is a technique that selectively retains certain duplicate chunks instead of removing them, thereby mitigating the severe locality disruption caused by their removal. However, rewriting sacrifices the deduplication ratio due to its conservative policy (evaluated in § 3.1). More fundamentally, it is a heuristic approach that does not resolve the core challenge.

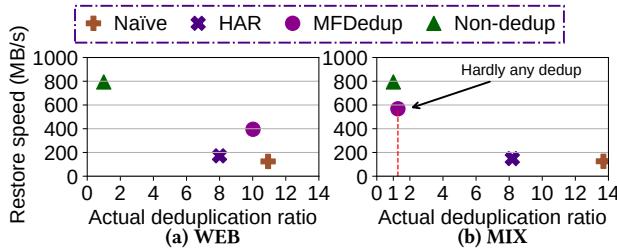
*Reordering* [53, 54] is a technique that globally reorganizes deduplicated chunks through data migration, which copies chunks and places them with optimized data layout, to mitigate fragmentation. However, current reordering approaches assume time-sequential backup images from identical backup sources. When they are applied to non-sequential images or images from different sources, their effectiveness drops dramatically (evaluated in § 3.1). This limitation stems from the inherent conflict between their migration principle and duplicate detection mechanisms (discussed in § 5.5).

## 2.4 GC in Deduplicated Backup Storage

Garbage collection (GC) reclaims storage by removing invalid chunks (no longer referenced by any backup). These chunks cannot be directly overwritten because containers are immutable. Invalid chunks are generated as a result of backup deletion: during deletion, the references between the deleted backup and its associated chunks are logically removed. However, such logical deletion defers the physical reclamation until the chunks become invalid. Since backup images are periodically replaced in a round-robin manner, invalid chunks accumulate, necessitating the periodic execution of GC.

The most commonly used method for GC is Mark-Sweep. It has two stages: the *mark* stage and the *sweep* stage. The mark stage generates *VC table* (e.g., Bloom filter or bitvector) that records all valid chunks [3, 10, 17, 46]. It also traverses every recipe of logically deleted backups to generate *GS list*, which records containers with invalid chunks. The sweep stage scans and checks all chunks in the containers of *GS list* one by one, and valid chunks of them (determined with the *VC table*) are copied to the new container [3, 10, 17, 46]. Existing works mainly accelerate the mark stage but lack research on the sweep stage (i.e., the migration).

Backup storage typically retains only recent backups. It periodically deletes outdated versions, which results in regular backup turnover and garbage collection. Prior studies have highlighted the substantial I/O overhead and high frequency of garbage collection [10, 27, 38]. A study from DELL EMC reveals that 21% of total data are reclaimed and ingested weekly within the backup workload from Data Domain products, indicating a significant turnover rate [38]. Furthermore, although backups are typically rotated in days or weeks, the median lifetime of a backup image in a commercial backup product is only 20 days [38]. These workloads make garbage collection a very heavy process and usually takes a long period (from several days to the duration of



**Figure 2.** The actual deduplication ratio and restoration performance of four approaches. MFDedup degrades to a non-dedup solution for datasets from multiple sources.

a backup cycle) to execute [10, 17, 34]. With garbage collection so time-consuming, concurrently backing up data leads to competition for disk I/O and processing resources, significantly affecting performance [10].

### 3 Observations and Motivation

#### 3.1 Limitations of Existing Approaches

As highlighted in § 2.3, fragmentation is an inherent issue in deduplication. Both existing approaches (*rewriting* [12, 25, 42] and *reordering* [53, 54]) have significant drawbacks.

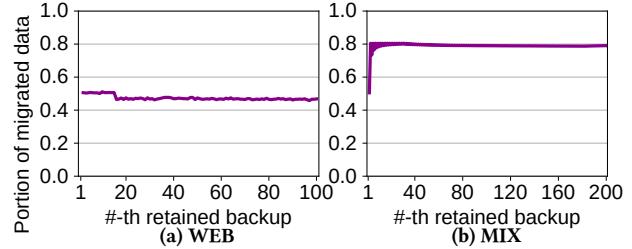
To analyze the effectiveness of these approaches, we select HAR [12], and MFDedup [54] as representative rewriting and reordering approaches, respectively. We compare them with a baseline (Non-dedup), that retains all duplicates, and a naive strategy (Naïve), which simply removes all duplicates. The evaluation is conducted on two datasets, WEB and MIX. WEB is composed of 100 snapshots of a news website from different days, and MIX mixes 100 snapshots of Redis databases with the snapshots of WEB.

The results (Fig. 2 and Fig. 3) reveal key limitations of these methods. Naïve achieves a high deduplication ratio but suffers from poor restoration performance. HAR mitigates fragmentation by tolerating some duplicates, improves restoration performance by 25% compared to Naïve, but at the cost of a significant reduction (up to 40%) of the deduplication ratio. MFDedup performs well on WEB, but requires costly data migration (50%–80% of the dataset, Fig. 3). Furthermore, on MIX, MFDedup fails to remove duplicates effectively (Fig. 2(b)), as it eliminates duplicates between adjacent backups only, which is ineffective when backups images are from multiple sources. The reason is discussed in § 5.5.

In summary, ① rewriting provides a direct trade-off between deduplication ratio and fragmentation, and achieves only marginal restoration speed improvements, while ② reordering improves restoration performance at the expense of overhead and limited applicability. Thus, we seek to develop a reordering-based approach that eliminates these drawbacks.

#### 3.2 Motivation: Piggyback Defragmentation on GC

Both *garbage collection* (GC) and *reordering-based defragmentation* are fundamental techniques in storage systems, commonly applied to disks [21, 30–32], file systems [18–



**Figure 3.** MFDedup causes a heavy data migration overhead in reorganizing deduplicated chunks. Portions are the ratio of migrated size to processed dataset size.

20, 23, 29, 41], and deduplicated storage [53, 54]. Although they are rarely discussed together, their purposes are, in fact, surprisingly similar. Specifically, reordering-based defragmentation leverages data migration to aggregate scattered chunks of files to improve the efficiency of data accessing, while GC leverages data migration to aggregate scattered valid chunks for space reclamation, as shown in Fig. 4 (a) and (b). Both address the issue of scattered data through data migration, which causes the main overhead for them.

This similarity opens an unexplored opportunity: allowing reordering-based defragmentation to capitalize on the existing data migration of GC to hide its own migration overhead, or, in other words, to *piggyback reordering-based defragmentation on garbage collection*. As shown in Fig. 4(c), we integrate reordering into the mark-sweep GC process, hiding its I/O operations within, effectively achieving defragmentation without additional overhead.

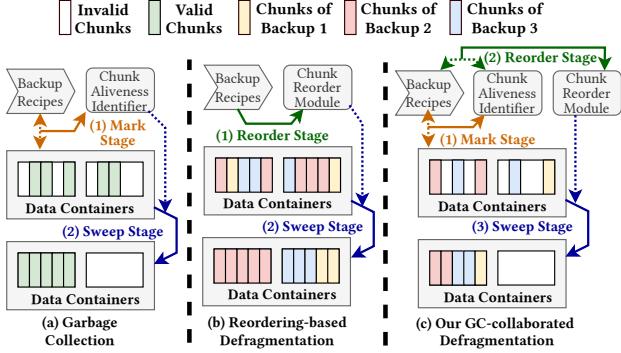
A key concern is whether GC occurs frequently enough for timely reordering-based defragmentation. Fortunately, backup storage systems typically exhibit frequent GC cycles (§ 2.4). This ensures that our approach is performed regularly.

#### 3.3 Challenges

Although piggybacking GC offers the opportunity to perform defragmentation on a deduplicated backup system, implementing it is nontrivial owing to two challenges:

*How to reduce fragmentation for all backups simultaneously?* Chunk sharing in deduplication presents a fundamental challenge to defragmentation: When two backups share common unique chunks, Integrating these chunks into one backup often results in their dispersion from another. Fig. 5 provides an example. In Data Layout 1, chunks for Backup4 are scattered, whereas the chunks for Backup1 are grouped together. However, if we aggregate chunks for Backup4, as seen in Data layout 2, the chunks of Backup1 become fragmented. Previous works [54] only propose a solution for very limited use cases (only for backups taken from the same backup source, such as the same virtual machine or database), and left this issue an open problem. Therefore, a new chunk layout is desired to handle fragmentation for all backups.

*How to reduce read amplification arising from granularity mismatch?* The granularity mismatch between fragmented



**Figure 4.** The workflows of GC and reordering-based defragmentation are similar, and our solution exploits this opportunity to achieve a collaborative approach.

backup data and containers introduces additional read amplification during restoration. Deduplicated backups are inherently fragmented, as they consist of multiple physically scattered pieces of varying sizes, each containing several chunks. Since the backup system stores data in fixed-sized containers, these fragmented pieces are often co-located with unrelated data within the same container, exacerbating read amplification during restoration. Therefore, an effective strategy is required to mitigate the impact of this mismatch and optimize read efficiency.

## 4 Design

We propose GCCDF, a unified garbage collection and defragmentation approach. It optimizes data migration of GC. During migration, GCCDF reorders the chunks online. This involves two key tasks: (1) determine which chunks should be put together during reordering, and (2) align the reordered chunks with containers while preserving locality.

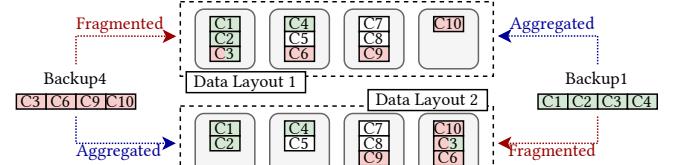
Recall that we mention two challenges in § 3.3. The two tasks aim to address these challenges, respectively, and we introduce two corresponding techniques: locality-promoting chunk clustering and container-adaptable cluster packing.

### 4.1 Locality-Promoting Chunk Clustering

Locality-promoting chunk clustering reorders chunks during data migration in garbage collection. The purpose is to promote locality across backups collectively, rather than to reduce it for one backup at the cost of exacerbating it for another (as illustrated in Fig. 5).

This technique stems from a fundamental question: How can we identify the *compatibility* among the locality of different backup images? Are there cases where certain chunks can be arranged to benefit all backups or, at the very least, avoid degrading their locality?

**The Way to Achieve Locality Compatibility.** We find that the *ownership* of chunks provides a way to achieve such compatibility. The ownership of a chunk is the set of backups that refer to it. If a chunk's ownership includes a backup, the chunk is essential for restoring the backup. Moreover, if all



**Figure 5.** The dilemma in defragmenting deduplicated chunks. Integrating chunks of one backup will scatter chunks of another one. C3 is a common chunk for two backups.

chunks within a container share the same ownership, then all of them are required when any backup of their ownership is restored; otherwise, none of them is needed.

Ownership reflects the access patterns of chunks, as it determines if a chunk is accessed on restoration of a backup. Grouping chunks with the same access patterns enhances the locality of backups and effectively reduces fragmentation. For example, as illustrated in Fig. 6, Data Layout 2 groups chunks with the same ownership:

- Chunks 1, 5, 7 are referenced by all three backups;
- Chunks 2, 4, 8 are referenced by Backup  $\alpha$  and  $\beta$ ;
- Chunks 3, 6, 9 are only referenced by Backup  $\alpha$ .

In this way, when we restore any backup, all chunks are either necessary or unnecessary. For example, to restore Backup  $\beta$ , chunks 2, 4, 8 are necessary, while 3, 6, 9 are not needed. Therefore, this layout ensures no read amplification during restoration, effectively mitigating fragmentation across all backups.

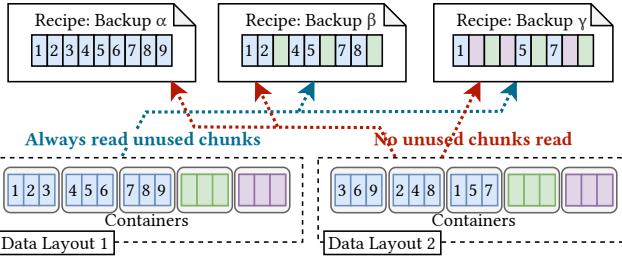
In contrast, Data Layout 1 is a negative example. It organizes chunks in their appearance order, which mixes chunks with different ownerships. For example, the container containing chunks 1, 2, 3 is friendly to backup 1, because they are necessary to restore Backup 1. However, it is unfriendly to Backup 2 and Backup 3, because it contains data chunks not needed, which causes read amplification during restoration.

**Clustering Strategy to Achieve Locality Compatibility.** Accordingly, we propose a locality-promoting chunk clustering, which manages chunks with the following steps:

- Gather chunk ownership information from the recipe of deduplicated backup images.
  - Learn all component chunks of each backup images from its recipe.
  - Analyze which backups refer to each chunk (i.e., ownership) in overall.
- Traverse all chunks, and classifies them into clusters based on their ownership.

By ensuring that chunks within the same cluster are always grouped while those from different clusters remain separate, locality compatibility across all backup images is preserved. In this way, the Challenge 1, reducing fragmentation for all backups simultaneously, is addressed.

**A Remaining Issue.** Our locality-promoting chunk clustering still suffers from misalignment. Data Layout 2 in Fig. 6 only illustrates a special case where the number of chunks



**Figure 6.** An example of different data layouts. *Data Layout 1* is the naturally generated by deduplication, and *Data Layout 2* groups chunks that are always used/unused together, which avoid read amplification on restoring all backups.

in each cluster precisely matches the container size. This matching is the reason why it works so well: The access pattern unit of chunks (i.e., clusters) and the I/O unit of backup storage (i.e., containers) are perfectly aligned.

However, in practical deployment, cluster sizes vary and rarely align exactly with container boundaries. This misalignment leads to chunks with different ownerships (also access patterns) being mixed within the same container (i.e., basic I/O unit), ultimately causing read amplification. Therefore, The following question is how to minimize the impact of misalignment between the variable-sized clusters and fixed-sized containers?

#### 4.2 Container-Adaptable Cluster Packing

While the previous technique (§ 4.1) focuses on clustering chunks, container-adaptable cluster packing addresses the organization and sequencing among clusters. It packs clusters with an arranged order, and fills them sequentially into the containers.

The primary challenge lies in minimizing the impact of mixing chunks with different ownerships within the same container. Such mixing is inevitable due to the misalignment between clusters and container boundaries. When restoring a backup, which happens to be in the difference set of ownerships, the system reads unused chunks, leading to read amplification.

This raises a critical question: Do the effects of mixing chunks from different clusters within a container remain consistent under all conditions? If not, how can we strategically determine which chunks to mix to minimize these effects?

**The Variable Impact of Chunk Mixing.** We find that the impact of such mixing varies depending on the similarity of the mixed chunks. The more similar the ownership of these chunks are, the lower the probability that mixing them will have an impact.

Consider a scenario with four backups, where, after deduplication, all their chunks are classified into three clusters based on their ownership:

- Cluster A: Chunks with ownership {Backups 1, 2, 3, 4}.
- Cluster B: Chunks with ownership {Backups 1, 3, 4}.

- Cluster C: Chunks with ownership {Backups 1, 2, 4}.

We analyze three cases of chunk mixing and their corresponding probabilities of suffering read amplification:

① Mixing chunks from Clusters A and B within a container: Restoring Backups 1, 3, and 4 does not cause read amplification since all mixed chunks are required. However, when restoring Backup 2, chunks from Cluster B are read unnecessarily. Therefore, this case leads to a 25% (1/4) probability of suffering read amplification, and the ownership similarity between these clusters is 75% (3 common backups out of 4).

② Mixing chunks from Clusters A and C within a container: Following the same reasoning, the probability of suffering read amplification remains 25% (1/4). The ownership similarity between these clusters is also 75%.

③ Mixing chunks from Clusters B and C within a container: Also following the same reasoning, the probability of suffering read amplification remains 50% (2/4) probability. The ownership similarity between these two clusters is 50% (2 common backups out of all 4 backups).

This analysis highlights that different mixing strategies lead to varying degrees of read amplification. Furthermore, it suggests that the probability of incurring an impact when mixing chunks is directly correlated with the ownership similarity of these chunks. The higher the similarity, the lower the likelihood of an impact.

However, an additional question arises: multiple clusters may have the same degree of ownership similarity between them. For multiple mixing choices with the same degree of similarity, which one should be preferred?

For this issue, we prioritize the cluster with the longest matching suffix in their ownership. The longest matching suffix refers to the longest sequence of backup indices that both clusters share at the end of their ownership lists. For instance, although the mixing cases ① (Clusters A and B) and ② (Clusters A and C) have the same ownership similarity, we prefer ① because the identical suffix of their ownership is longer (Backups 3 and 4). This preference is driven by two key reasons: (1) Deduplication tends to increase fragmentation in later backups since they are more likely to reference duplicate chunks. Therefore, we prioritize reducing read amplification for these backups. (2) Later backups persist through more backup turnover cycles, making locality optimizations for them more effective over a longer period.

**Packing Strategy for Optimized Mixing.** Based on the above considerations for minimizing the impact of mixing chunks from different clusters, we propose container-adaptable cluster packing, which follows these steps:

- Gather clusters generated by the previous technique (§ 4.1) and maintain a list to record the output sequence.
- Identify the cluster with the largest ownership size and place it at the beginning of the list as the initial entry.
- The algorithm then enters an iterative process:

- Based on the cluster at the end of the list, find its most similar cluster in terms of ownership. If multiple candidate clusters have the same similarity, prioritize the one with the longest matching suffix in ownerships.
- Append the selected cluster to the end of the list and proceed to the next iteration.
- Repeat the iteration until all clusters are placed in the list, which serves as the final sequence for packing.

With this list, a rearranged chunk sequence is prepared for reordering chunks during the data migration in garbage collection. This chunk sequence is directly derived by sequentially retrieving clusters from the list and extracting chunks from each cluster in order. Locality-compatible chunk clustering ensures that adjacent chunks share the same ownership, while container-adaptable cluster packing ensures that adjacent clusters have the most similar ownerships.

## 5 Implementation

### 5.1 Overview

Leveraging the two techniques proposed in § 4, we implement GCCDF as illustrated in Fig. 7. The workflow of GCCDF is integrated into the garbage collection process and is invoked between the mark and sweep stages.

The mark stage scans containers with reclaimable space and identifies valid chunks within them. Following this, GCCDF operates through three sequential modules:

- Preprocessor: Load all valid chunks into memory and collect reference information for ownership analysis.
  - Analyzer: Perform locality-promoting chunk clustering and generate clusters based on chunks' ownership.
  - Planner: Execute container-adaptable cluster packing and determine the final reordered sequence of migrated chunks.
- Subsequently, during the sweep stage, GC relocates the valid chunks to new containers according to the reordered sequence, while the old containers are discarded. Finally, we eliminate invalid data and mitigate data fragmentation simultaneously.

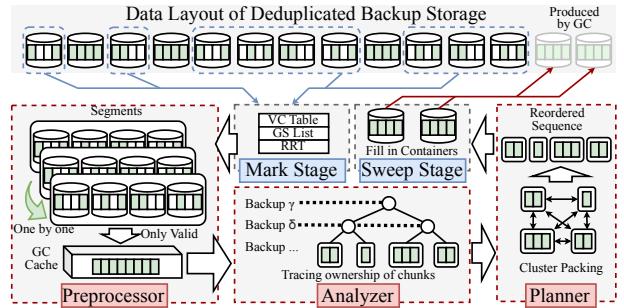
### 5.2 Preprocessor

The Preprocessor bridges the gap between the GC mark stage and the Analyzer. Its primary function is to identify and prepare valid chunks for reordering.

The workflow of the Preprocessor is illustrated in Fig. 8. Generally, it has three main tasks: ① Segment GC-involved containers for efficient processing. Subsequent operations run separately on each segment; ② Identify valid chunks within the current segment and load them into a memory cache; ③ Collect the reference relationship between scanned containers and backups, which is used for analyzing ownership of chunks.

In the following, we introduce these tasks, respectively.

**Segmentation.** The Preprocessor first groups GC-involved containers into segments. All the following processes in GC-CDF to reordering chunks run on each segment separately.



**Figure 7.** An overview of GCCDF framework (red-shaded modules) illustrating how it works in mark-sweep GC (blue-shaded modules). White/blue chunks are invalid/valid ones.

Segmentation is introduced with practical considerations in mind: ① Chunk reordering also faces fragmentation. It usually gathers scattered but logically sequential chunks and writes them back, leads to inefficient I/O, similar to restoration. ② To mitigate this inefficiency, we read all to-be-reordered chunks, cache them in memory, and write them back after reordering. ③ However, caching all these chunks is impractical due to memory constraints. Thus, we adopt segmentation so that reordering is executed on a smaller set of chunks at once, balancing efficiency, cost, and effectiveness. This trade-off is further discussed in § 5.5.

The size of the segments is a configurable parameter, and the impact of it is evaluated in § 6.5.

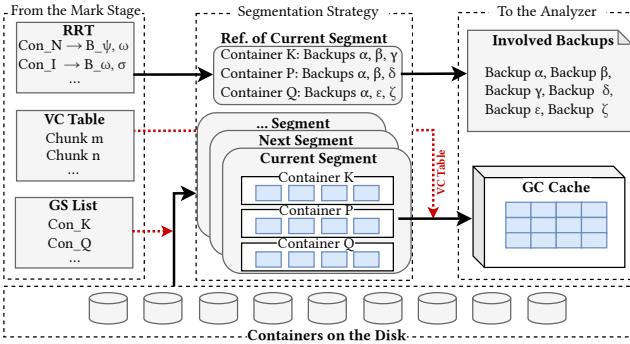
Upon generating segments with feasible granularity, the Preprocessor along with subsequent modules handle them sequentially through iterative loops.

**Identifying and Caching Valid Chunks, and Collecting Their Ownerships.** For each segment, the Preprocessor identifies valid chunks, reads them into a memory cache, and collects the reference information for containers in the current segment. These chunks and information are prepared for the Analyzer and Planner.

Specifically, the Preprocessor begins by traversing all of the chunks in the current segment and checks whether it is a valid chunk with the VC table, which is provided by the mark stage. Invalid chunks are ignored, and for valid chunks, the Preprocessor stores them in a *GC Cache*, which is a memory cache to maintain chunks in the current segment. Because the Preprocessor only works on one segment at a time, the maximum size of the GC Cache is limited.

Then, the Preprocessor collects the reference information of the containers within the current segment from the hash table *RRT*, which is also provided by the mark stage. For example, if the current segment contains Containers K, P, Q, the Preprocessor searches for the corresponding entries, and combines them to generate a list *Involved Backups*, as shown in Fig. 8. These reference information assist the Analyzer to reduce the complexity in analyzing the ownership of chunks.

Subsequently, the Preprocessor sends *Involved Backups*



**Figure 8.** The workflow of the Preprocessor. It separates the containers scheduled for GC into segments, and initiates defragmentation on each segment.

to the Analyzer, waits for chunks in the current segment written with reordered sequence (after the Planner and the final sweep stage of GC), and then proceeds to handle the next segment.

### 5.3 Analyzer

The Analyzer is responsible for clustering valid chunks based on their ownerships, ensuring that chunks with similar access patterns are grouped together. It operates on the output of the Preprocessor. The key task of the Analyzer is to determine the ownership of chunks, namely identifying the backups that refer to each chunk.

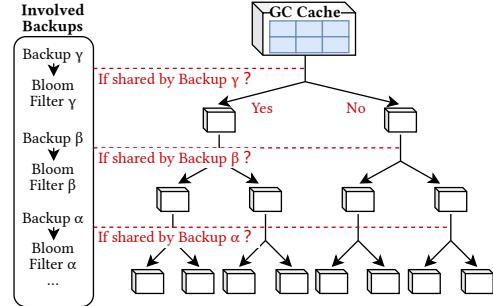
**Core Mechanism.** As illustrated in Fig. 9, the core mechanism of the Analyzer is implemented as follows:

1. The Analyzer gathers all chunks in the current segment and checks whether they are referenced by the first backup. Based on the results, the chunks are split into two groups.
2. For each group, the Analyzer then checks whether the chunks are referenced by the second backup, further splitting them into subgroups, resulting in four groups in total.
3. This process repeats iteratively, checking each subgroup against successive backups until all chunks are classified according to their references against all backups.

During the mechanism, we utilize a binary tree. First, all chunks in a segment are in a single node. Every time a backup is checked, each leaf node derives two child nodes. Each chunk of the leaf node is put into the right child if it is referenced by the backup and the left child if not. This corresponds to group splitting of the core mechanism. The derivation repeats until all backups are checked. Each leaf node of the final binary tree contains chunks in the same group. Obviously, chunks in each group have the same ownership, and chunks from different groups differ in their ownerships.

**Optimizations.** Upon this mechanism, the Analyzer introduces several optimizations.

① *Efficiently reference checking for chunks.* During each round of reference checking, directly using backup recipes for reference lookup is inefficient. A recipe is a sequential list of records specifying the chunks used by each backup. Deter-



**Figure 9.** An example of clustering chunks in the Analyzer by tracing which backups share them.

mining whether a chunk is referenced by a backup requires scanning every record, making the process time-consuming. To address this, the Analyzer constructs bloom filters for recipes of *Involved Backups* using chunk fingerprints as keys. It significantly reduces the time of determining ownership.

② *A reversed order of reference checking across backups.* The backups are checked in reverse to determine their reference to each chunk. Specifically, the most recent backup is checked first. Then, the second most recent backup is checked. This process repeats until the earliest backup. This is for the convenience of container-adaptable cluster packing run by the Planner, which is detailed in the next subsection.

③ *Denying inefficient group splitting.* In extreme cases, the number of leaf nodes (corresponding to distinct ownerships) may become excessive. Notably, for  $n$  backups, the number of leaf nodes in the binary tree can reach up to  $2^n$ . In such scenario, there are a excessive number of leaf nodes may be generated, each containing only a few chunks. Such clustering is overly fragmented. To mitigate this, we introduce a threshold of splitting. If a leaf node is small enough, it is prevented from further splitting.

④ *An efficient data structure for leaf nodes for fast traversal.* Each leaf node has pointers to neighboring nodes, thus forming a doubly linked list. This can be used by the Planner to traverse leaf nodes efficiently. In addition, the leaf nodes store only chunk pointers rather than the actual data for fast and efficient splitting.

### 5.4 Planner

The Planner determines the optimized sequence for migrated chunks during GC. To achieve this, it first executes container-adaptable cluster packing to rearrange the clusters generated by the Analyzer. Then, it sequentially traverses the rearranged clusters, extracting chunks in order to generate the final reordered chunk sequence.

**Workflow.** As illustrated in Fig. 10, the workflow of the Analyzer is implemented with assistance of the binary tree in the Analyzer module. Steps are listed as follows:

1. Traversing the leaf nodes in the binary tree from left to right. As discussed in the optimizations in § 5.3, the Analyzer maintains a doubly linked list for the leaf nodes.

- This linked list enables efficient traversal of the leaf nodes.
2. Reading chunk fingerprints from the leaf nodes and inserting them into the *Migration Order* table.
  3. Subsequently, the Planner constructs the final reordered chunk sequence, according to the sequence specified in the *Migration Order* table.

After generating the reordered chunk sequence for the current segment, the sweep stage in GC is executed. This sweep stage retrieves the required chunks from the GC cache and fills them into containers with the reordered sequence, completing the GC process for the current segment. Once all segments have been processed, the garbage collection procedure is completed.

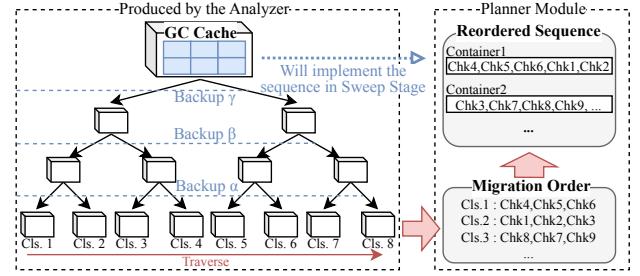
**A Binary-Tree-Assisted Implementation of Container-Adaptable Cluster Packing.** The above workflow does not appear to include any step for container-adaptable cluster packing. This is because the container-adaptable cluster packing mechanism is integrated into the design of the binary tree in the Analyzer.

For example, during the traversal of leaf nodes in the binary tree (from left to right), a similarity-based order is implicitly achieved. It is because the construction method of the binary tree naturally ensures that the clusters corresponding to adjacent leaf nodes in the binary tree exhibit similar ownership. This property stems directly from the round-based reference checking, where leaf nodes are formed based on whether their chunks are referenced by specific backups. As illustrated in Fig. 10, Cls.1 and Cls.2 contain chunks shared by similar backups: chunks in Cls.1 are referenced by Backups  $\gamma$ ,  $\beta$ , and  $\alpha$ , while chunks in Cls.2 are referenced by Backups  $\gamma$ , and  $\beta$ . At the same time, the higher the level of the binary tree, the more ancestor nodes the leaf nodes share, and the higher their similarity between leaf nodes.

Moreover, as discussed in the optimizations of § 5.3, we deliberately impose a reverse order when checking chunk references during binary tree construction. This design choice is based on the observation that adjacent leaf nodes typically share the most common ancestor nodes. Consequently, during the binary tree construction, chunks from adjacent leaf nodes should yield identical reference check results against the corresponding backup of their respective ancestors. By imposing this reverse order, we ensure that adjacent leaf nodes exhibit consistent reference patterns for the most recent backups, since they correspond to the nearest common ancestors. This design effectively prioritizes the locality of the most recent backups.

## 5.5 Discussion

**On Generating RRT in the Mark Stage of GC.** RRT can be efficiently generated during the mark stage, as detailed in § 2.4. We first require the mark stage to generate the *GS list* (containers scheduled for GC) and then the *VC table* (valid chunks). Since constructing the *VC table* involves traversing all alive backup recipes [11], RRT can be built simultaneously.



**Figure 10.** An example of the Planner producing a chunk reordering plan (continue the example in Fig. 9).

During traversal, we log which backup recipes reference chunks in containers from the *GS list*. For example, if chunks in Container\_K are referenced by Backups  $\alpha$ ,  $\beta$ , and  $\gamma$ , we record  $\langle K \rightarrow \alpha, \beta, \gamma \rangle$  in *RRT*.

The size of *RRT* remains manageable. In an extreme case where each container in the *GS list* maps to 100 recipes (a deduplication ratio of nearly 100:1), each entry is about 800 bytes (assuming 8 bytes per recipe ID). Given a typical container size of 4MB, the size ratio is about 5000:1. Even with 5TB of GC-needed containers, *RRT* would require approximately 1GB, which is feasible.

**The Size of the Binary Tree.** The size of the binary tree depends on the number of nodes, which store chunk fingerprints. Assuming a segment consists of 100 containers (4MB each) with an average chunk size of 4KB, it contains around 80,000 chunks. In the worst case, each node holds one chunk. With a 20-byte SHA1 fingerprint and an 80-byte node structure, the total tree size per segment is about 8MB. Our evaluation shows that a segment's binary tree has 1200–1600 leaf nodes and is only about 1.6MB. Moreover, the tree can be released immediately after processing the segment.

**The Trade-offs in Configuring the Size of Segments.** The concept of a segment is integral to the architecture of GCCDF. The choice of segment size significantly influences GCCDF's efficiency in several aspects. E.g., utilizing a larger segment size can improve defragmentation efficiency, which reducing the chance that chunks needing to be merged are located in different segments. However, it concurrently escalates the GC cache's memory overhead (it caches all containers in a segment), increases the Analyzer's complexity (much more chunk sharing cases), and enlarges the binary tree (the number of levels in the tree relies on the involved backups in a segment). At the same time, introducing segments also offers the opportunity for parallel defragmentation, since the workflows for defragmenting each segment are totally independent. This issue is studied in § 6.5.

**Can GCCDF Leverage Versioning Information from COW File Systems in Primary Storage?** Copy-on-Write (COW) file systems inherently track data modifications, enabling versioning and differencing. This raises the question: Can GCCDF utilize COW features to enhance chunk-sharing

detection for backup grouping?

Despite its potential, fundamental layout differences prevent such integration. Primary storage uses fixed-sized blocks (e.g., 4KB), whereas backup storage relies on *content-defined chunking* (CDC) [28, 39, 45] to use variable-sized chunks to detect duplicates effectively. This mismatch creates an information gap: COW tracks block-level changes, while GCCDF requires chunk-level data sharing insights.

The purpose of CDC and variable-sized chunks is to address the *boundary shift problem*, where fixed-sized chunking fails to deduplicate shifted content after small modifications. CDC anchors chunk boundaries to content patterns, ensuring stable duplicates identification. For instance, if backup  $v_1$  has chunks {A,B,C}, and  $v_2$  modifies B to B', CDC retains alignment for A and C, unlike fixed-sized chunking.

Due to these fundamental differences in granularity and tracking, COW metadata is insufficient for GCCDF chunk-sharing analysis. Thus, GCCDF must operate at the CDC chunk level rather than reconciling incompatible block-based versioning.

**Why GCCDF could Handle Many More Use Cases than Previous Defragmentation Approaches.** Mitigating the fragmentation caused by deduplication is a challenging task, as discussed in Section 3.3. A key difficulty arises from the large and unmanageable number of clusters (i.e., ownership of chunks), as illustrated in Fig. 9: In a  $n$ -level binary tree, the number of leaf nodes can reach  $2^{n-1}$ , making managing these clusters highly complex.

Existing work, such as MFDedup, attempts to address this issue by restricting chunk sharing to only consecutive backups. Specifically, if Backups  $\alpha$ ,  $\beta$ , and  $\gamma$  are stored sequentially and Chunk M appears in both Backups  $\alpha$  and  $\gamma$  (but not in  $\beta$ ), MFDedup is unable to eliminate redundancy and instead stores two replicas of Chunk M. This simplification reduces the maximum number of clusters (about  $n \times (n - 1)/2$ ), making clustering and reordering feasible. However, it also limits MFDedup's applicability in real-world backup scenarios, such as those involving backup images from multiple users, as shown in Fig. 2 (b).

In contrast, GCCDF employs a different approach to reduce the number of clusters. First, our segmentation strategy limits chunk sharing analysis to each segment, thereby minimizing the likelihood of encountering complex cases. Second, the Analyzer introduces a binary tree to trace cluster derivations, enabling monitoring and avoidance of fragmented clusters. As a result, GCCDF adapts effectively to diverse backup workloads while mitigating fragmentation more efficiently.

## 6 Evaluation

### 6.1 Evaluation Setup

**Testbed.** We perform experiments on a server with an Intel Xeon Platinum 8468V CPU, 128GB memory, two Intel S4610

**Table 1.** Four datasets used in the evaluation.

Name	Original Size	Descriptions
WIKI	1.2 TB	120 snapshots, each is a full backup of a specific language Wikipedia at a particular point in time.
CODE	394 GB	220 versions of source codes from Chromium, LLVM, and Linux kernel project.
MIX	809 GB	200 snapshots from a news website and a Redis database from different days.
SYN	1.1 TB	240 volumes of synthetic backups by simulating file create/delete/modify operations [36].

2TB SSDs and an Intel P4610 1.6TB SSD. The server runs Ubuntu 22.04.5 LTS.

**Configurations and Approaches.** We implemented a prototype integrating multiple rewriting techniques in the deduplication workflow and GCCDF in garbage collection.

FastCDC [45] is used for chunking (1 KB min, 4 KB avg, 32 KB max); SHA-1 for fingerprinting; and a 4 MB container size, following prior works [4, 12, 44, 48, 49]. GCCDF supports variable segment sizes (default 100 containers). We use six approaches for comparison:

- **Naïve.** A naive approach that runs typical Mark-Sweep garbage collection without any restore acceleration.
- **Capping [25]/HAR [12]/SMR [42].** Typical rewriting techniques with typical garbage collection.
- **MFDedup [54].** Typical reordering approach for deduplication. A dedicated migration stage is required.
- **GCCDF.** Our approach, which runs collaboratively with garbage collection.

**Datasets.** The evaluated datasets, listed in Table 1, are widely used in prior works [6, 40, 43, 47, 54]. Three are real-world datasets, covering Wikipedia, code repository, and a mix of Redis and website snapshots. One is synthetic, generated based on a study of realistic backup storage [36].

**Methodology.** For evaluation, we utilize the P4610 SSD for the user storage, while a RAID array composed of two S4610 SSDs serves as the backup storage.

**Major Evaluations:** When backing up images, we read the images from user storage, perform deduplication, and then store remaining data in the backup storage (similar to previous studies [4, 25, 42, 49]). The backup storage always retains the 100 most recent backups, probabilistically deletes the earliest 20 backups (as the turnover rate in § 2.4) in each round, and then runs GC to reclaim storage space. After all rounds of ingestion, deletion, and GC, the remaining backups are restored to test restoration speed. The evaluation focuses on the bottleneck on the backup storage side.

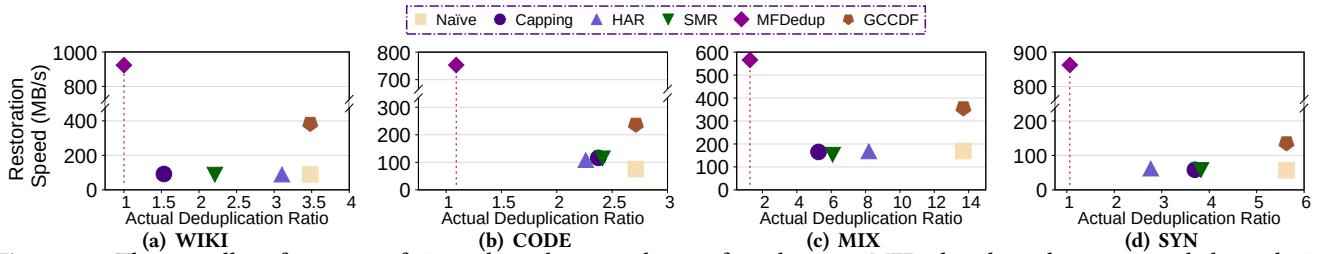
Our evaluation is conducted around the following aspects:

**1. Deduplication Ratio and Restoration Performance (§ 6.2):** How effective are these methods in preserving the deduplication ratio and improving restoration performance?

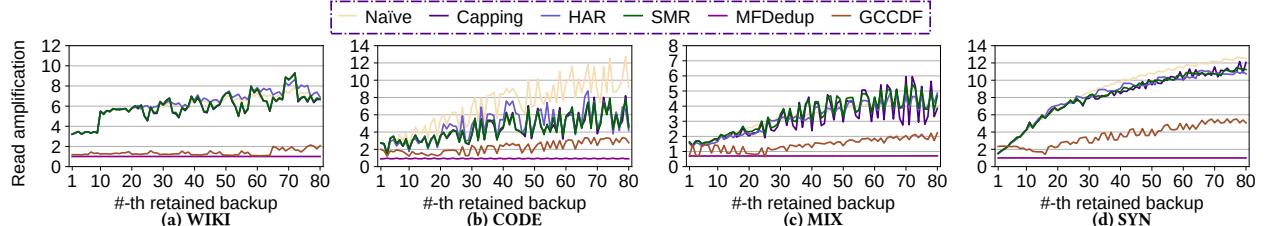
**2. The Effectiveness of Mitigating Fragmentation (§ 6.3):** To what extent do these methods reduce fragmentation, specifically in terms of the read amplification?

**3. The I/O Overhead and Time Cost of GC (§ 6.4):** How does GCCDF impact the I/O overhead and time cost in GC?

**4. Sensitivity Analysis of Designs in GCCDF (§ 6.5):**



**Figure 11.** The overall performance of six evaluated approaches on four datasets. MFDedup degrades to a non-dedup solution and thus is free from fragmentation. GCCDF performs well on both sides. Up and to the right is better.



**Figure 12.** The read amplification in restoring retained backups, which reflects the degree of fragmentation. GCCDF effectively handles the fragmentation caused by deduplication. Lower is better.

How do our clustering and packing strategies impact defragmentation effectiveness and GC efficiency?

## 6.2 Overall Performance

This subsection studies the overall performance of all six approaches evaluated in four datasets, and Fig. 11 suggests the results. The actual deduplication ratio is computed by  $\frac{\text{The Original Size of the Dataset}}{\text{The Actual Space Cost after Deduplication}}$ , and the restoration performance is calculated by  $\frac{\text{The Size of a Backup}}{\text{The time cost to restore the Backup}}$ .

MFDedup removes hardly any duplicates in these datasets (as good as no deduplication), each of which is comprised of backups from various sources. Rewriting approaches (Capping, HAR, and SMR) achieve little acceleration in restoration speed, but suffer a significant deduplication ratio loss in most cases. Specifically, SMR, a rewriting technique recently proposed, achieves a marginally improved restoration performance, outperforming Naïve by factors of  $1.01 \times$  (WIKI),  $1.5 \times$  (CODE),  $1.02 \times$  (MIX), and  $1.03 \times$  (SYN). Nevertheless, it leads to decreases of the deduplication ratio of 37% (WIKI), 11% (CODE), 56% (MIX), and 33% (SYN). This suggests that rewriting is not effective in terms of budget savings as a lower deduplication ratio leads to higher storage space and disk costs, and the budget could have been utilized to construct a RAID with more disks or invest in higher-performing disks can naturally enhance restoration performance. Therefore, preserving the deduplication ratio is key for an anti-fragmentation approach to be effective.

Generally, GCCDF always achieves  $2.7 \times$  (WIKI),  $3.1 \times$  (CODE),  $2.1 \times$  (MIX),  $2.3 \times$  (SYN) higher restoration performance than Naïve, while preserving the same deduplication ratio. On the one hand, it shows the effectiveness of GCCDF in dealing with fragmentation, and its proposed principles actually work, including clustering chunks shared by the same back-

ups, and merging similar clusters to adapt to the container-based data layout. On the other hand, it is because GCCDF never tolerates any duplicate chunks; instead, it mitigates fragmentation by reorganizing chunks and avoids *robbing Peter to pay Paul* which occurs in rewriting.

## 6.3 Quantitative Assessment of Fragmentation

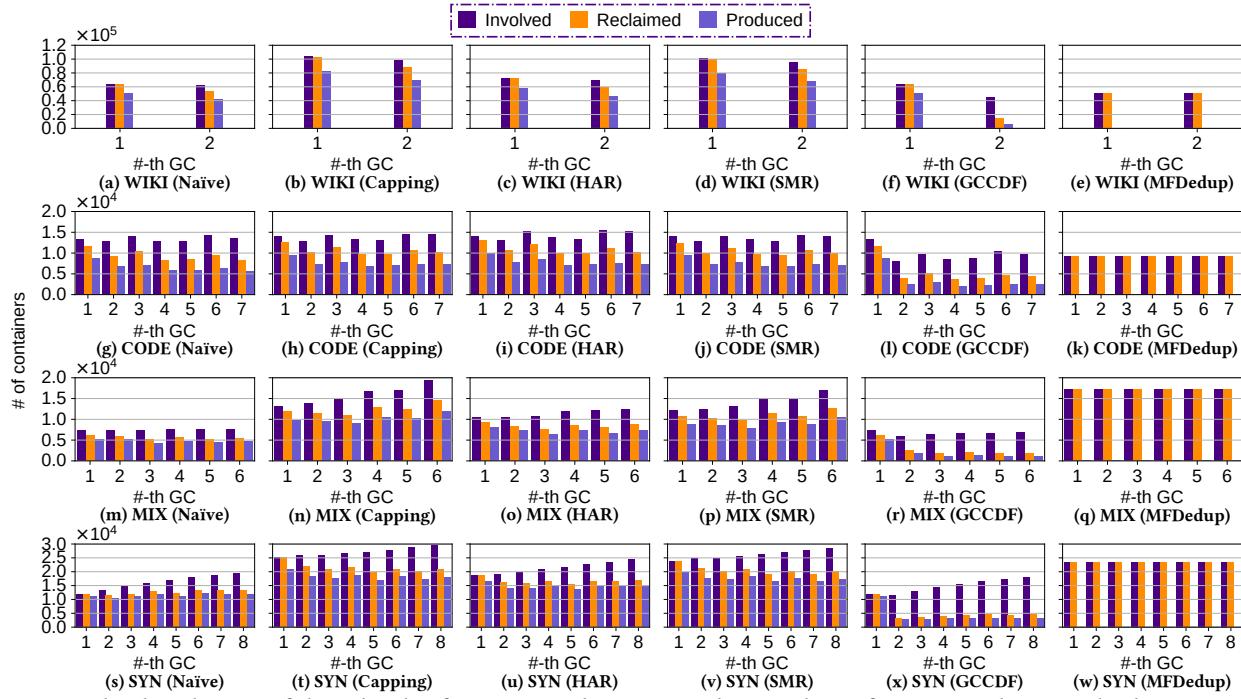
The previous subsection shows the benefits of GCCDF in terms of backup restoration performance. This subsection studies the degree of fragmentation in the deduplicated data after these approaches are applied, and delves into the factors contributing to GCCDF’s rapid restoration.

As mentioned in § 2.3, fragmentation during the restoration of backup images primarily results in read amplification. Therefore, studying the degree of read amplification after applying different approaches could demonstrate their effectiveness in mitigating fragmentation.

Fig. 12 suggests the read amplification factor when restoring each left backup. The read amplification factor of a backup is calculated by  $\frac{\text{The Size of Read Containers in Restoration}}{\text{The Size of Restored Backup}}$ .

The evaluation always retains the recent 100 backups, and stops in the final round, after deleting 20 old backups, running GC, and waiting for the 20 new backups. Therefore, the number of the remaining backups is 80.

It is obviously that GCCDF always achieves the smallest read amplification factors, except for MFDedup (it has no fragmentation due to no chunk sharing and no deduplication benefits). Specifically, GCCDF only exhibits average read amplification factors of  $1.3 \times$  (WIKI),  $2.2 \times$  (CODE),  $1.4 \times$  (MIX), and  $3.6 \times$  (SYN). In contrast, SMR, a rewriting technique recently proposed, experiences average read amplification factors of  $5.9 \times$  (WIKI),  $4.3 \times$  (CODE),  $3.4 \times$  (MIX), and  $8.2 \times$  (SYN). MFDedup maintains minimal read amplification as it fails to remove duplicates, thus preventing fragmentation.



**Figure 13.** The distribution of three kinds of containers during GC when applying five approaches. *Involved* containers hold chunks referenced by deleted backups and may have invalid chunks. *Reclaimed* containers, confirmed to have invalid chunks, need to be reclaimed after copying forward valid chunks. *Produced* containers receive valid chunks during GC. Less is better.

The results show that GCCDF successfully mitigates the most fragmentation without sacrificing any deduplication ratio, and provides a good solution for the dilemma between backup budget (i.e., deduplication ratio) and disaster recovery (i.e., restoration performance).

#### 6.4 Garbage Collection Overhead with GCCDF

As mentioned in § 2.4, GC first builds the list of containers that contain invalids, and then traverses these containers. Finally, it copies forward the valid chunks to new containers and deletes old ones. GCCDF is involved in copying forward where it reorders the migrated chunks. In this subsection, we consider the impact of GCCDF from two perspectives.

The number of rounds of garbage collection in evaluated datasets varies due to the differing number of backups. For instance, in the WIKI dataset, garbage collection is executed only twice because it contains just 120 backups while we retain 100 backups and rotate 20 backups in each round. The situation is similar for the other datasets.

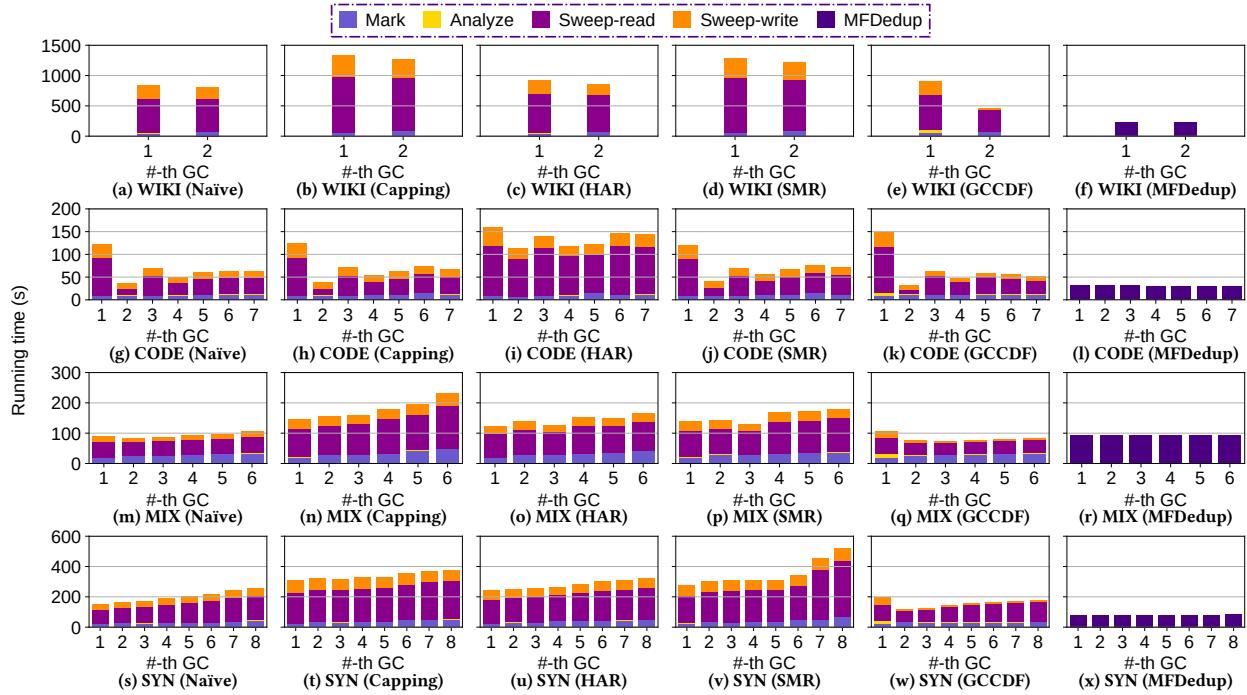
**Data Migration Scale During GC.** First, we analyze how different approaches influence the scale of data migration during GC. This study focuses on the distribution of three kinds of containers during GC, including involved containers, reclaimed containers, and produced containers. Involved containers include chunks that are referenced by the deleted backups, and thus potentially have invalid chunks. Reclaimed containers are verified to have invalid chunks, and need to be reclaimed after copying forward valid chunks. Produced containers are generated by receiving valid chunks during

GC. This distribution suggests the I/O overhead in data migration during GC, which is reported as the main cost in garbage collection [10]. Fig. 13 shows the distribution.

In general, rewriting methods such as Capping, HAR, and SMR incorporate a greater variety of all three kinds of containers compared to Naïve, which means larger-scale data migration and I/O overhead. This is because these strategies accommodate duplicates, resulting in a higher number of invalid chunks to reclaim. Conversely, GCCDF significantly decreases all three kinds of containers after the initial round of GC. Specifically, the number of produced containers in GC-CDF is almost just 1/3 of that in other approaches, and thus the write I/O during GC is also hugely reduced. This is because GC also suffers from the fragmentation issue (scattered backups are always more difficult to reclaim than aggregated ones), and GCCDF hugely mitigates fragmentation, and thus also benefits the efficiency of GC.

Note that as MF DEDUP lacks the container-based setting, the number of containers for this approach is derived by dividing the total data processed during GC by the container size for easy comparison. MF DEDUP fails to remove duplicates, thus it has more but aggregated invalid chunks, and it does not produce any new container during GC.

**Breakdown of GC Time Cost.** Second, we study the end-to-end time cost breakdown of garbage collection, including the mark stage, analyzing stage, sweep-read stage, sweep-write stage. Except for the analyzing stage only existing in GCCDF, other three stages are common for all approaches.



**Figure 14.** The time cost breakdown of garbage collection when applying five approaches on four evaluated datasets. GCCDF even accelerates garbage collection in most cases. Less is better.

Fig. 14 shows the results.

All approaches have nearly the same time cost in the *Mark* stage. This stage works on recipes to identify involved containers and a table of valid chunks, and nearly has no differences between the evaluated approaches. The *Analyze* stage is unique for GCCDF, which reorganizes migrated chunks, but only takes a small proportion of the total garbage collection, which suggests the efficiency of the Analyzer and the Planner in GCCDF. The *sweep-read* and *sweep-write* stages are both parts of data migration, in which the *sweep-read* stage reads all reclaimed containers, and the *sweep-read* stage writes all produced containers. GCCDF achieves shorter sweep-read and sweep-write stages in most cases, because its reclaimed containers and produced containers are smaller than those of other approaches, as in Fig. 13.

MFDedup avoids mark-sweep GC by directly deleting aggregated invalid chunks, incurring time costs only for deletion. However, unlike other methods that remove only unique chunks (not shared) of deleted backups, MFDedup handles more data due to its inability to remove duplicates and share chunks.

Generally, although GCCDF incurs some extra computational overhead for reorganizing migrated chunks, it offsets this by decreasing I/O overhead during GC, as storage systems typically exhibit sensitivity to I/O.

## 6.5 Sensitivity Analysis of Designs

In this section, we study the impact of segment sizes in our *locality-compatible chunk clustering*, and the effectiveness of our *container-adaptable cluster packing*. Due to page limita-

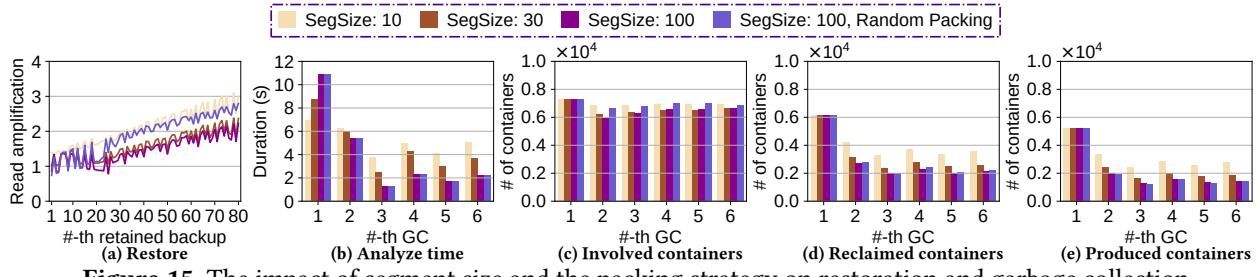
tion, we only illustrate the results on the MIX dataset, and those on other datasets are similar.

Fig. 15 suggests the results when GCCDF is configured with different segment sizes and a random packing strategy. SegSize # means that each segment includes # containers, which leads to a memory cost for the GC cache of # × 4MB.

Fig. 15 (a) shows the read amplification of restoring backups when applying different segment sizes and the random packing strategy. Here we show the read amplification instead of the restoration performance just for a clear suggestion of the differences. The results suggest that an overly small segment size hinders effective defragmentation and causes larger read amplification. This is because a small segment size causes fewer chunks for each cluster, which leads to the production of new fragments when aligning the clusters with the container-based storage structure.

Besides, when applying the random packing strategy instead of our proposed one, the read amplification dramatically increase (about 20% on average), suggesting the effectiveness of our recency-prioritized cluster packing strategy.

Additionally, the defragmentation effectiveness under different segment sizes also impacts the efficiency of garbage collection. Fig. 15 (c), (d), and (e) show the number of involved, reclaimed, and produced containers during GC with different segment sizes. Generally, a larger segment size leads to a better defragmentation effectiveness, and decreases the GC workloads (i.e., involved, reclaimed, and produced containers), because chunks of the discarded backups are well aggregated. Instead, a smaller segment size means less defrag-



**Figure 15.** The impact of segment size and the packing strategy on restoration and garbage collection.

mentation effectiveness, and thus leads to more containers need to be reclaimed. Besides, the random packing strategy has little impact on the critical migration overhead of GC, as in Fig. 15 (d), (e), although it slows restoration.

Fig. 15 (b) suggests the time cost of GCCDF during GC, when applying different segment sizes. For the first round of GC, although all configurations have the same containers to reclaim (Fig. 15 (d)), GCCDF with a larger segment size takes more time to reorganize chunks due to a large number of chunk sharing cases to analyze. However, for the following round of GC, GCCDF with a smaller segment size costs more time in reorganizing chunks instead. This is because in the following GC round, GCCDF with a smaller segment size has more containers to reclaim (Fig. 15 (d)).

## 7 Conclusion

We introduce GCCDF, an innovative approach that piggybacks defragmentation on GC to amortize data movement overhead. By handling the conflicts of locality between backups and the granularity mismatch between chunk clustering and storage units, GCCDF effectively mitigates fragmentation while maintaining high deduplication ratios. Comprehensive evaluations, simulating long-term backup system operations, demonstrate GCCDF’s superior performance compared to existing reordering and rewriting methods. GCCDF offers a groundbreaking solution to fragmentation issues in deduplicated backup systems.

## 8 Acknowledgments

We sincerely appreciate the anonymous reviewers for their insightful comments and valuable feedback. We are especially grateful to our shepherd, Dr. Marc Shapiro, for his dedication and guidance. His multiple rounds of detailed and constructive suggestions significantly improved our work. This work was supported in part by the National Natural Science Foundation of China under Grant 62472127, Guang-Dong Basic and Applied Basic Research Foundation under Grant 2023A1515110072, and the Shenzhen Science and Technology Program under Grants GXWD20231128111309001.

## References

- [1] Yamini Allu, Fred Douglass, 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 *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC ’15)*.
- [3] Fabiano C. Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. 2013. Memory efficient sanitization of a deduplicated storage system. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST ’13)*. 81–94.
- [4] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. 2019. Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST ’19)*. 129–142.
- [5] Geyao Cheng, Lailong Luo, Junxu Xia, Deke Guo, and Yuchen Sun. 2023. When Deduplication Meets Migration: An Efficient and Adaptive Strategy in Distributed Storage Systems. *IEEE Transactions on Parallel and Distributed Systems* 34, 10 (2023), 2749–2766.
- [6] Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, and Zhongjie Wu. 2021. Coupling Right-Provisioned Cold Storage Data Centers with Deduplication. In *Proceedings of the 50th International Conference on Parallel Processing (ICPP ’21)*. 17:1–17:11.
- [7] John Rydning David Reinsel, John Gantz. 2018. The Digitization of the World From Edge to Core. *IDC White Paper* (2018), 1–28.
- [8] Biplob Debnath, Sudipta Sengupta, and Jin Li. 2010. ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX ATC ’10)*.
- [9] Cai Deng, Xiangyu Zou, Qi Chen, Bo Tang, and Wen Xia. 2024. The Design of a Lossless Deduplication Scheme to Eliminate Fine-Grained Redundancy for JPEG Image Storage Systems. *IEEE Trans. Comput.* 73, 5 (2024), 1385–1399.
- [10] Fred Douglass, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano C. Botelho. 2017. The Logic of Physical Garbage Collection in Deduplicating Storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST ’17)*. 29–44.
- [11] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. 2019. Data Domain Cloud Tier: Backup here, backup there, deduplicated everywhere!. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC ’19)*. 647–660.
- [12] 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 Distributed Systems* 27, 3 (2016), 855–868.
- [13] John Gilmore, Jay Fenlon, et al. 1992. GNU tar: an archiver tool. (1992).
- [14] Phil Goodwin. 2021. The State of Data Protection and Disaster Recovery Readiness: 2021. *IDC White Paper* (2021), 1–12.
- [15] Phil Goodwin. 2022. The State of Ransomware and Disaster Preparedness: 2022. *IDC White Paper* (2022), 1–12.
- [16] Phil Goodwin. 2024. The State of Disaster Recovery and Cyber-

- Recovery, 2024–2025: Factoring in AI. *IDC White Paper* (2024), 1–21.
- [17] Fanglu Guo and Petros Efstathopoulos. 2011. Building a High-performance Deduplication System. In *Proceedings of 2011 USENIX Annual Technical Conference (USENIX ATC '11)*. 1–14.
- [18] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. 2017. Improving File System Performance of Mobile Storage Systems Using a Decoupled Defragmenter. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. 759–771.
- [19] Cheng Ji, Li-Pin Chang, Sangwook Shane Hahn, Sungjin Lee, Riwei Pan, Liang Shi, Jihong Kim, and Chun Jason Xue. 2019. File Fragmentation in Mobile Devices: Measurement, Evaluation, and Treatment. *IEEE Transactions on Mobile Computing* 18, 9 (2019), 2062–2076.
- [20] Cheng Ji, Li-Pin Chang, Liang Shi, Chao Wu, Qiao Li, and Chun Jason Xue. 2016. An Empirical Study of File-System Fragmentation in Mobile Storage Systems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '16)*.
- [21] Yuhun Jun, Shin-Hyun Park, Jeong-Uk Kang, Sang-Hoon Kim, and Euiseong Seo. 2024. We Ain't Afraid of No File Fragmentation: Causes and Prevention of Its Performance Impact on Modern Flash SSDs. In *Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST '24)*. 193–208.
- [22] Jürgen Kaiser, André Brinkmann, Tim Süß, and Dirk Meister. 2015. Deriving and comparing deduplication techniques using a model-based classification. In *Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15)*. 11:1–11:13.
- [23] Jianjun Li, Yuhui Deng, Jiande Huang, Yi Zhou, Qifan Yang, and Geyong Min. 2025. Gecko: Efficient Sliding Window Aggregation With Granular-Based Bulk Eviction Over Big Data Streams. *IEEE Transactions on Knowledge & Data Engineering* 37, 2 (2025), 698–709.
- [24] Jingwei Li, Zuoru Yang, Yanjing Ren, Patrick P. C. Lee, and Xiaosong Zhang. 2020. Balancing storage efficiency and data confidentiality with tunable encrypted deduplication. In *Proceedings of the Fifteenth EuroSys Conference (EuroSys '20)*. 22:1–22:15.
- [25] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. 2013. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '13)*. 183–198.
- [26] 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 *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*.
- [27] Dutch T Meyer and William J Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage (TOS)* 7, 4 (2012), 1–20.
- [28] Fan Ni and Song Jiang. 2019. RapidCDC: Leveraging duplicate locality to accelerate chunking in CDC-based deduplication systems. In *Proceedings of the ACM symposium on cloud computing (SOCC '19)*. 220–232.
- [29] Yanqi Pan, Hao Huang, Yifeng Zhang, Wen Xia, Xiangyu Zou, and Cai Deng. 2024. Delaying Crash Consistency for Building A High-Performance Persistent Memory File System. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 43, 9 (2024), 2620–2634.
- [30] Shujie Pang, Yuhui Deng, Zhaorui Wu, Genxiong Zhang, Jie Li, and Xiao Qin. 2025. RDA: A Read-Request Driven Adaptive Allocation Scheme for Improving SSD Performance. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 44, 2 (2025), 416–429.
- [31] Jonggyu Park and Young Ik Eom. 2021. FragPicker: A New Defragmentation Tool for Modern Storage Devices. In *Proceedings of the 28th Symposium on Operating Systems Principles (SOSP '21)*. 280–294.
- [32] Wenjie Qi, Zhipeng Tan, Jicheng Shao, Lihua Yang, and Yang Xiao. 2022. InDef: An Advanced Defragmenter Supporting Migration Offloading on ZNS SSD. In *Proceedings of the 40th International Conference on Computer Design (ICCD '22)*. 307–314.
- [33] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*. 714–729.
- [34] Dilip Nijagal Simha, Maohua Lu, and Tziccker Chiueh. 2013. A scalable deduplication and garbage collection engine for incremental backup. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13)*. Article 16, 12 pages.
- [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 (2024), 23:1–23:30.
- [36] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. 2012. Generating Realistic Datasets for Deduplication Analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*. 261–272.
- [37] Veritas. 2024. Purpose-Built Backup Appliance (PBBA): The Ultimate Guide. *Veritas White Paper* (2024), 1–10.
- [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 *Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '12)*.
- [39] Binzheshuo Wan, Lifeng Pu, Xiangyu Zou, Shiyi Li, Peng Wang, and Wen Xia. 2022. Supercdc: A hybrid design of high-performance content-defined chunking for fast deduplication. In *2022 IEEE 40th International Conference on Computer Design (ICCD '22)*. IEEE, 170–178.
- [40] 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 (2022).
- [41] Donglei Wu, Weihao Yang, Xiangyu Zou, Hao Feng, Dingwen Tao, Shiyi Li, Wen Xia, and Binxing Fang. 2024. BIRD+: Design of a Lightweight Communication Compressor for Resource-Constrained Distribution Learning Platforms. *IEEE Transactions on Parallel Distributed Systems* 35, 11 (2024), 2193–2207.
- [42] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. 2019. Improving Restore Performance in Deduplication Systems via a Cost-Efficient Rewriting Scheme. *IEEE Transactions on Parallel and Distributed Systems* 30, 1 (2019), 119–132.
- [43] 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. *Proceedings of IEEE* 104, 9 (2016), 1681–1710.
- [44] 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 *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '15)*.
- [45] 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 *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC '16)*. 101–114.
- [46] Jingsong Yuan, Xiangyu Zou, Han Xu, Zhichao Cao, Shiyi Li, Wen Xia, Peng Wang, and Li Chen. 2022. A Focused Garbage Collection Approach for Primary Deduplicated Storage with Low Memory Overhead. In *Proceedings of the 40th International Conference on Computer Design (ICCD '22)*. 315–323.
- [47] 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 *Proceedings of the 2023 USENIX Annual Technical Conference (USENIX ATC '23)*. 133–148.
- [48] 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 *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*. 121–128.
- [49] 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 Distributed System* 31, 10 (2020), 2302–2314.
- [50] Nannan Zhao, Muhui Lin, Hadeel Albahar, Arnab K. Paul, Zhijie Huan, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Ali Anwar, and Ali Raza Butt. 2024. An End-to-end High-performance Deduplication Scheme for Docker Registries and Docker Container Storage Systems. *ACM Transactions on Storage* 20, 3 (2024), 18.
- [51] Benjamin Zhu, Kai Li, and R. Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*. 269–282.
- [52] Benjamin Zhu, Kai Li, and R. Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System.. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST' 08)*, Vol. 8. 1–14.
- [53] 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 *Proceedings of 2022 USENIX Annual Technical Conference (USENIX ATC '22)*. 19–36.
- [54] 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 *Proceedings of the 19th USENIX Conference on File and Storage Technologies (FAST '21)*. 171–185.

## A Artifact Appendix

### A.1 Abstract

This artifact appendix serves as a self-contained guide for evaluating the GCCDF artifact. It provides detailed descriptions of the hardware, software, and configuration requirements necessary to reproduce the experimental results presented in the paper. The artifact consists of a prototype backup system with deduplication capabilities, implementing various rewriting and garbage collection strategies to support the paper's claims.

To facilitate reproducibility, the document outlines the execution steps for different experimental configurations, including multiple deduplication and garbage collection techniques. The artifact's compilation and execution processes are described, along with the dependencies required for proper functionality. Additionally, a cloud-based experimental environment has been set up to enable remote access for evaluators. The document also includes instructions for running experiments, logging results, and generating plots to validate the findings.

By linking key claims from the paper to corresponding experimental procedures, this appendix ensures that evaluators can systematically verify the artifact's performance and conclusions.

### A.2 Description & Requirements

The artifact of GCCDF primarily consists of a prototype backup system with deduplication functionality, which implements various rewrite strategies and garbage collection strategies to meet the experimental requirements described in the paper.

Specifically, the differences between the methods are as follows:

#### Deduplication Workflow:

Ingesting datasets → Chunking → Calculating fingerprints → Identifying duplicate chunks (**where rewriting techniques work**) → Writing unique chunks.

When the number of stored backups reaches 100, old images are evicted. Specifically, 20 images are deleted, and garbage collection is performed to free up storage space. New backup images are then written. **GCCDF runs alongside garbage collection.**

Therefore, the different methods described in the paper can be implemented by configuring the artifact as follows:

- **Naive:** No rewriting technique with naive garbage collection;
- **Capping:** Capping rewriting technique with naive garbage collection;
- **SMR:** SMR rewriting technique with naive garbage collection;
- **HAR:** HAR rewriting technique with naive garbage collection;
- **GCCDF:** No rewriting technique with GCCDF-powered garbage collection.

The MFDedup method compared in the paper can be obtained at <https://github.com/HIT-HSSL/MFDedup>. Its workflow and background assumptions are significantly different, making it difficult to integrate into the same artifact.

### A.2.1 How to access

<https://github.com/Borelset/GCCDF>, also DOI:10.5281/zenodo.14926415

### A.2.2 Hardware dependencies

The artifact requires the following Hardware:

- Backup Space: A RAID-0 array consisting of two Intel S4610 drives.
- Dataset Space: An Intel P4610 drive.
- Memory: larger than 64GB
- CPU: Intel(R) Xeon(R) Platinum 8468V

### A.2.3 Software dependencies

The artifact requires the following dependencies:

- `isal_crypto`: [https://github.com/intel/isa-l\\_crypto](https://github.com/intel/isa-l_crypto)
- `jemalloc`: <https://github.com/jemalloc/jemalloc>
- `openssl`
- `zstd`: <https://github.com/facebook/zstd>

### A.2.4 Benchmarks

The artifact uses the following datasets:

- **WIKI**: 120 snapshots, each is a full backup of a specific language Wikipedia at a particular point in time.
- **CODE**: 220 versions of source codes from Chromium, LLVM, and Linux kernel project.
- **MIX**: 200 snapshots from a news website and a Redis database from different days.
- **SYN**: 240 volumes of synthetic backups by simulating file create/delete/modify operations

### A.3 Set-up

The artifact is organized using CMake, so it can be compiled directly with the following commands:

```
1 cd build
2 cmake ..
3 make
```

### A.4 Evaluation workflow

For evaluation, we utilize the P4610 SSD for the user storage, while a RAID array composed of two S4610 SSDs serves as the backup storage. When backing up images, we read the images from user storage, perform deduplication, and then store remaining data in the backup storage. The backup storage always retains the 100 most recent backups, probabilistically deletes the earliest 20 backups in each round, and then runs GC to reclaim storage space. After all rounds of ingestion, deletion, and GC, the remaining backups are restored to test restoration speed. The evaluation focuses on the bottleneck on the backup storage side.

#### A.4.1 Major Claims

- GCCDF improves the restoration speed by 2.1× compared to the state-of-the-art method, SMR, without

- GCCDF offers a significant advantage over MFDedup by achieving a  $6.45\times$  higher deduplication ratio in typical scenarios.
- GCCDF dramatically reduces the I/O overhead of GC, since it aggregates scattered chunks of deleted backups, improving reclamation efficiency.

#### A.4.2 End-to-end Experiments

1-2 days for CODE and MIX dataset, and 2-3 days for WIKI and SYN datasets.

[How to]

Ingesting all backup images of a dataset, and then restoring the retained ones. Image retention and garbage collection will occur automatically during ingestion.

We focus on the read amplification in the restoration and the cost in garbage collection.

### *[Execution]*

The artifact can be executed using the following shell scripts:

- ## 1. Ingesting Backup Images.

```

1 ChunkPath= /path/for/deduplicated/chunks
2 LogicPath= /path/for/recipes/of/ingested/images
3 BatchFile= /path/to/dataset/list
4 RestorePath= /path/to/restoration/path/
5 LCGC=true # whether enable GCCDF
6 EnableRewriting=false # whether enable rewriting
7 Rwriting=capping # which rewriting technique
8 CappingThres=20 # parameter for capping
9 SegSize=100 # parameter for GCCDF

10
11 LOGPATH=/path/for/running/log
12 RUNPATH=/path/for/executable

13
14 echo "running $BatchFile"
15
16 $RUNPATH --task=batch --BatchFilePath=$BatchFile
    ↵ --ChunkFilePath=$ChunkPath
    ↵ --LogicFilePath=$LogicPath
    ↵ --RestorePath=$RestorePath --LCbasedGC=$LCGC
    ↵ --RewritingMethod=$Rwriting
    ↵ --CappingThreshold=$CappingThres
    ↵ --enable_rewriting=$EnableRewriting
    ↵ --GCSegmentSize=$SegSize >
    ↵ $LOGPATH/dedup.log
2. Restoring Backup Images.

1 ChunkPath=/path/for/deduplicated/chunks
2 LogicPath=/path/for/recipes/of/ingested/images
3 BatchFile=/path/to/dataset/list
4 RestorePath=/path/to/restoration/path/
5 LCGC=true # whether enable GCCDF
6 EnableRewriting=false # whether enable rewriting

```

```
7 Rwriting=capping # which rewriting technique
8 CappingThres=20 # parameter for capping
9 SegSize=100 # parameter for GCCDF
10
11 LOGPATH=/path/for/running/log
12 RUNPATH=/path/for/executable
13
14 for ((i=0; i < $2; i++)); do
15     printf -v paddedi "%03d" $i # Pad the
16     # number to 3 digits
17     echo "Restoring # $paddedi"
18     echo 3 > /proc/sys/vm/drop_caches
19     $RUNPATH --task=get
20     #--ChunkFilePath=$ChunkPath
21     #--LogicFilePath=$LogicPath
22     #--RestorePath=$RestorePath --RecipeID=$i
23     #> $LOGPATH/restore_${paddedi}.log
24 done
```

## [Results]

The key performance indicators that the experiment focuses on, including read amplification, GC time cost, and migrated containers in GC, have been marked in the log. These indicators can be used to compare different methods.