Building a High-performance Fine-grained Deduplication Framework for Backup Storage with High Deduplication Ratio

Xiangyu Zou and Wen Xia, *Harbin Institute of Technology, Shenzhen;* Philip Shilane, *Dell Technologies;* Haijun Zhang and Xuan Wang, *Harbin Institute of Technology, Shenzhen*

https://www.usenix.org/conference/atc22/presentation/zou

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Building a High-performance Fine-grained Deduplication Framework for Backup Storage with High Deduplication Ratio

Xiangyu Zou†*, Wen Xia*†*, Philip Shilane*∗*, Haijun Zhang*†*, and Xuan Wang*†
† Harbin Institute of Technology, Shenzhen ∗ Dell Technologies
Corresponding author: xiawen@hit.edu.cn

Abstract

Fine-grained deduplication, which first removes identical chunks and then eliminates redundancies between similar but non-identical chunks (i.e., delta compression), could exploit workloads' compressibility to achieve a very high deduplication ratio but suffers from poor backup/restore performance. This makes it not as popular as chunk-level deduplication thus far. This is because allowing workloads to share more references among similar chunks further reduces spatial/temporal locality, causes more I/O overhead, and leads to worse backup/restore performance.

In this paper, we address issues for different forms of poor locality with several techniques, and propose MeGA, which achieves backup and restore speed close to chunk-level deduplication while preserving fine-grained deduplication's significant deduplication ratio advantage. Specifically, MeGA applies ① a backup-workflow-oriented delta selector to address poor locality when reading base chunks, and ② a delta-friendly data layout and "Always-Forward-Reference" traversing in the restore workflow to deal with the poor spatial/temporal locality of deduplicated data.

Evaluations on four datasets show that MeGA achieves a better performance than other fine-grained deduplication approaches. In particular, compared with the traditional greedy approach, MeGA achieves a 4.47–34.45× higher backup performance and a 30–105× higher restore performance while maintaining a very high deduplication ratio.

1 Introduction

Chunk-level deduplication [2, 7, 18, 20, 27, 28, 40, 45, 54] has been widely used in backup storage systems to reduce storage costs, but it is limited by its coarse-grained processing granularity (i.e., file/chunk level) and can not completely exploit data workloads' compressibility. To achieve a higher deduplication ratio, fine-grained deduplication [22, 38, 47] is proposed.

Fine-grained deduplication, sometimes previously called "delta compression," not only focuses on duplicate chunks but also removes sub-chunk-level redundancies existing in similar but non-identical chunks, and it has been studied in several use cases [15, 37, 38, 51]. Typically, fine-grained deduplication first deduplicates identical chunks, then finds similar base chunks (among non-duplicates), and finally runs delta encoding between the new and base chunks to only store their differences (a.k.a., delta chunks) for space-saving. As a result, fine-grained deduplication could achieve a much higher deduplication ratio than chunk-level deduplication [37]. We use the term fine-grained deduplication, though some previous literature uses the term delta compression to refer to this entire process.

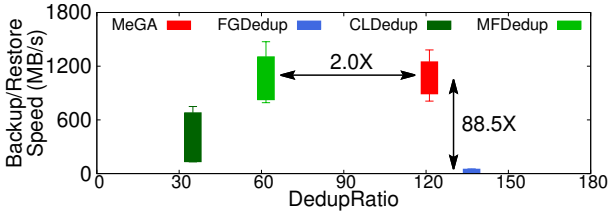


Figure 1: Performance of MeGA (our approach), FGDedup (a typical fine-grained deduplication approach similar to SDC [53]), CLDedup (a typical chunk-level deduplication approach [23]), and MFDedup (a special chunk-level deduplication approach [58]) on a website snapshot dataset.

However, fine-grained deduplication's performance is usually much worse than that of chunk-level deduplication because of further reducing data locality. Chunk-level deduplication usually suffers from the poor locality of deduplicated data, which has been mentioned in several previous works [12, 23, 58]. For example, when deduplicating a workload, we only store unique chunks and "share" chunks that appear in stored workloads as duplicates. Because chunks are stored in chronological order, this kind of "sharing" results in duplicate chunks and other unique chunks of this workload being scattered across the storage media, which leads to poor performance when restoring this workload. This problem is aggravated by fine-grained deduplication. It is because fine-grained deduplication introduces delta compression to exploit more compressibility among workloads, so workloads "share"

more data, decreasing locality, increasing I/O overheads, and leading to worse backup/restore performance.

Generally, different forms of the poor locality caused by delta compression impact backup and restore workflows. In the backup workflow, reading base chunks for delta encoding suffers from poor locality of base chunks (denoted by **Reading Base Issue**). Specifically, this issue is related to local compression, since consecutive chunks are compressed and must be decompressed together, which makes the compression unit become the I/O unit [6, 25, 44]. Thus, we have to read a compression unit even when only one or a few base chunks are needed, which leads to huge I/O amplification. In the restore workflow, the **Fragmentation Issue** [12, 23, 53] (that also exists in chunk-level deduplication) is exacerbated by the more complex dependencies in fine-grained deduplicated data. It is caused by a new kind of reference relationship between delta and base chunks, and this new kind of reference relationship further breaks spatial locality in fine-grained deduplicated data. Meanwhile, additional reference relationships between delta and base chunks also lead to a poor temporal locality in fine-grained deduplicated data. During delta decoding, base chunks and delta chunks must both be read, unlike restoring deduplicated data that only requires a single I/O read for a needed chunk, which makes the restore workflow repeatedly access containers to gather delta-base pairs (denoted by the **Repeatedly Accessing Issue**).

In this paper, we aim to improve these locality issues based on several observations and techniques.

For the *Reading Base Issue*, we apply a backup-workflow-oriented delta selector to improve the efficiency of reading base chunks in the backup workflow. It is based on an observation that most base chunks are located in a few containers (e.g., 64.1% containers only include 8.31% of the base chunks when running a backup workflow in a studied dataset). According to this observation, our delta selector skips delta compression when base chunks are located in those "base-sparse containers". Without reading these "inefficient" containers, the efficiency of reading base chunks will be improved.

For the *Fragmentation Issue*, we propose a delta-friendly data layout, which covers the two-level reference relationships in fine-grained deduplicated data: the chunks–workloads reference relationship (also exists in chunk-level deduplication) and the additional delta–base reference relationship (caused by delta compression). The delta-friendly data layout handles the new dependencies and improves the spatial locality in fine-grained deduplicated data.

For the *Repeatedly Accessing Issue*, we observe the existence of "Always-Forward-Reference" traversing. It is a special path to traverse restore-involved containers, in which delta chunks always appear before their base chunks. By using this feature and exploiting the asymmetry of the I/O characteristics of storage media, we design a delta prewriting mechanism to deal with the poor temporal locality in deduplicated data, which first prewrites delta chunks to their location in the to-be-restored workload and then reloads them for decoding when later accessing their base chunks.

We propose MeGA, a fine-grained deduplication framework, by using the above techniques to address the *Reading Base Issue*, *Fragmentation Issue*, and *Repeatedly Accessing Issue*. As shown in Fig. 1, MeGA achieves performance close to chunk-level deduplication while preserving fine-grained deduplication's significant deduplication ratio advantage. The contributions of this paper are threefold:

- We analyzed several forms of poor locality caused by fine-grained deduplication, which leads to additional I/O overhead and poor backup/restore performance.
- We proposed techniques (i.e., the backup-workflow-oriented delta selector, the delta-friendly data layout, the "Always-Forward-Reference" traversing, and delta prewriting) to deal with these different issues caused by the poor locality.
- We proposed MeGA with these techniques to achieve performance close to chunk-level deduplication while preserving fine-grained deduplication's significant deduplication ratio advantage. Especially, compared with the traditional greedy approach [53], MeGA achieves a 4.47–34.45× higher backup performance and a 30–105× higher restore performance, while maintaining a very high deduplication ratio.

2 Background and Related Works

2.1 Fine-grained Deduplication

Fine-grained deduplication [10, 15, 37, 38, 44, 51] could achieve a much higher deduplication ratio than deduplication alone [9, 11, 17, 19, 21, 32–34, 39]. It focuses on redundancies not only between duplicate chunks but also between similar but non-identical chunks, and finally achieves sub-chunk-level detection as well as byte/string-level elimination.

However, fine-grained deduplication achieves a higher deduplication ratio while introducing additional computation and I/O overhead when applying delta compression between similar chunks. To address these challenges, many previous works have been proposed, and the additional computation overhead has been hugely reduced. For example, Zhang et al. [52] and Zou et al. [57] proposed much faster sketch methods by exploiting the locality in backup streams and content-based sampling, respectively. MacDonald [26] proposed Xdelta for fast delta encoding. Xia et al. [48, 49] and Tan et al. [41] presented chunking-inspired methods to further improve delta encoding/decoding speeds. Zhang et al. [53] extended the rewriting techniques [12, 23] from chunk-level deduplication to fine-grained deduplication to reduce the additional I/O overhead only in fine-grained deduplication's restore workflow.

Fine-grained deduplication has been employed in many other works. Xu et al. [50] introduced fine-grained deduplication for databases to reduce storage cost. Jain et al. [16] applied the idea of fine-grained deduplication in replica syn-

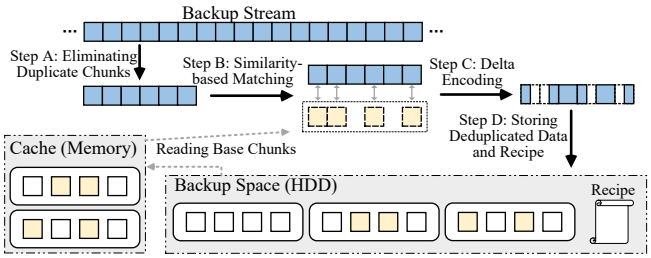


Figure 2: The backup workflow of fine-grained deduplication.

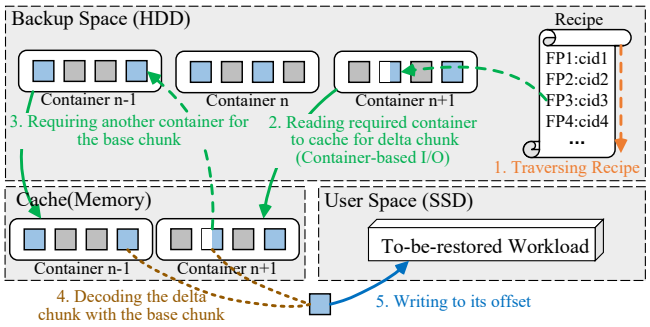
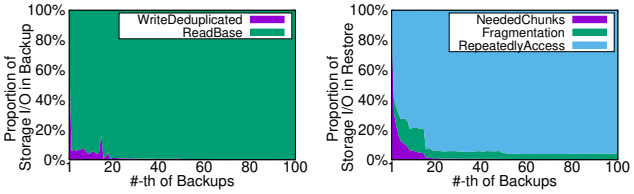


Figure 3: Restoring a delta chunk in the restore workflow of fine-grained deduplication.



(a) I/O in Backup workflow (b) I/O in Restore workflow

Figure 4: I/O overheads in backup and restore workflow.

chronization. Pucha et al. [35], Mogul et al. [30], and Zhou et al. [55] designed a detection mechanism for p2p system, which finds both identical and similar sources to accelerate downloads.

Fig. 2 shows a standard workflow for fine-grained deduplication: ① Split backup streams into chunks and calculate a fingerprint for each chunk. ② Check and eliminate duplicate chunks using the fingerprint index. ③ Calculate each unique chunk's sketches. Super Feature [5,22,24,37] is a typical kind of sketch. It first generates multiple local-sensitive hashes with rolling hashes and linear transformations, and then packs these local-sensitive hashes together into fewer Super Features to detect highly similar chunks. ④ Find similar candidates for unique chunks using a sketch index or cache. ⑤ If a similar candidate exists, read it as a base chunk, and delta encode the incoming chunk relative to the base, often generating a much smaller delta chunk. ⑥ All deduplicated chunks are stored in containers in order, and then each container will be compressed. ⑦ Generate a recipe for a backup stream by recording fingerprints of all needed chunks, including indirectly referenced base chunks.

2.2 Backup Workloads

In backup storage systems [29], workloads usually are a series of backups (i.e., successive snapshots of the primary data), and consecutive backups are usually similar, which has been reported and exploited in many existing studies [13,47,52]. Thus, due to the highly redundant nature of the data, backup storage often leverages data deduplication to greatly reduce the size of backups and save hardware costs.

Deduplicated data (i.e., chunks) are usually locally compressed and stored in immutable and fixed-size containers (e.g., 4MB). Containers are compatible with striping across multiple drives in a RAID configuration, and writing in large units achieves the maximum sequential throughput [23].

3 Observation and Motivation

3.1 Challenges

Fine-grained deduplication obtains a higher deduplication ratio than chunk-level deduplication with much worse backup/restore performance, but further fragments data locality. As mentioned in several previous works [12,23,58], chunk-level deduplication usually suffers from poor locality because chunks from a workload that are logically consecutive may refer to previously written chunks scattered across the disks. However, fine-grained deduplication has more serious locality issues. Specifically, fine-grained deduplication eliminates redundancies among similar chunks by creating more references to previously written chunks, which increases fragmentation. Meanwhile, this observation also means that as more space is saved, locality becomes worse. Poor locality harms both backup and restore performance.

Fig. 2 demonstrates the poor locality involved in reading base chunks for fine-grainded deduplication. Specifically, this issue is related to the local compression, since consecutive chunks must be decompressed entirely according to the compression unit (that also becomes the I/O unit) [6,25,44], which could be containers or compression regions (i.e., containers' sub-unit). Therefore, reading a compression unit when only one or a few base chunks are needed leads to I/O amplification. Generally, a larger I/O unit (e.g., containers) may cause a larger I/O amplification, but it also could opportunistically prefetch more base chunks and reduce costly random accesses on HDDs (due to locality of backup stream [38,52]). Even with a smaller I/O unit (e.g., 128KB container regions), reading bases remains a bottleneck [38]. Though it may cause less I/O amplification, reading some base chunks having locality with a small I/O unit can be disrupted by write tasks and result in more random seeks, because the backup workflow of fine-grained deduplication mixes reads and writes (i.e., reading base chunks and writing deduplicated data). Thus, we learn **Challenge 1**: Poor locality in the backup workflow causes inefficient I/O when reading base chunks.

In the restore workflow (like Fig. 3), there are two challenges. The first challenge in the restore workflow is the fragmentation problem, which is caused by poor spatial locality in

deduplicated data. It also exists in chunk-level deduplication, but it becomes more serious in fine-grained deduplication. It is because fine-grained deduplication allows workloads to share more similar chunks, but it also produces more references to previously written chunks. Therefore, fine-grained deduplication introduces another kind of reference relationship (i.e., between the base and delta chunks) and no longer only has one kind of reference relationship (i.e., between chunks and workloads). This makes the fragmentation problem more complex since the dependencies of each workload are distributed more widely. Thus, there exists **Challenge 2**: Delta-base relationships lead to more complex fragmentation problems than deduplication alone. The restore workflow also has **Challenge 3**: Delta-base dependencies cause poor temporal locality during delta decoding and causes repeated container reads. Without fine-grained deduplication, individual chunks can be read as needed to restore a file, but for fine-grained deduplication, base chunks and delta chunks must both be read. When chunks in a container are used (for unique or base chunks) across long time intervals, the restore workflow needs to alternately and repeatedly access containers to gather delta-base pairs for delta decoding.

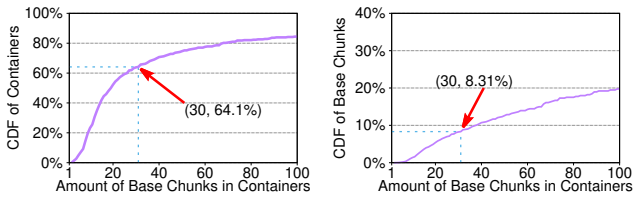
Finally, Fig. 4 suggests the seriousness of these challenges. It studies the I/O overheads of a basic fine-grained deduplication system with container I/O when backing up and restoring backup workloads from a WEB dataset (detailed in §5.1), which consists of 100 snapshots of a website. "WriteDeduplicatedData" means I/O for writing deduplicated data in the backup workflow, and "NeededChunks" means I/O for reading needed chunks. "ReadBase", "Fragmentation", and "RepeatedlyAccess" map to the above three challenges, respectively. We learn that these three challenges cause huge I/O overheads, and even "WriteDeduplicated" and "NeededChunks" only take about 0.3% and 1.12% of the total I/O in backup and restore workflows.

3.2 Selective Delta Compression

As Challenge 1 mentioned, poor locality in reading base chunks causes large I/O overheads in the backup workflow.

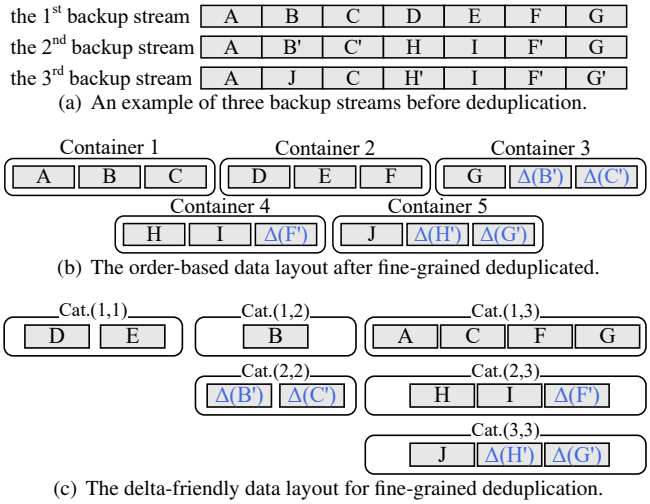
To this end, we studied datasets and observed that base chunks are not distributed evenly. For example, Fig. 5 gives the distribution of base chunks when backing up the 100^{th} backup in the WEB dataset. Fig. 5(a) suggests that 64.1% of containers include fewer than 30 base chunks, and Fig. 5(b) demonstrates that these containers only hold 8.31% of the total base chunks. We call these containers "base-sparse containers". Though there are only a few base chunks in these base-sparse containers, when requiring base chunks in one of them, we have to load the whole container from the disk, which causes a significant read amplification.

Thus, these observations motivate us to design a **backup-workflow-oriented delta selector**, which skips delta compression whose base chunks are located in "base-sparse containers" to avoid reading these "inefficient" containers. Thus,



(a) 64.1% of containers contain only ~30 base chunks. (b) These 64.1% containers only includes 8.31% of the total base chunks.

Figure 5: Base chunks are not distributed evenly.



(a) An example of three backup streams before deduplication.

(b) The order-based data layout after fine-grained deduplicated.

(c) The delta-friendly data layout for fine-grained deduplication.

Figure 6: An example of the order-based data layout versus the delta-friendly data layout.

it could reduce the I/O overheads in the backup workflow, and finally greatly improve the backup speed in fine-grained deduplication, which will be evaluated in §5.2.

3.3 Delta-friendly Data Layout

For Challenge 2, we use the example in Fig. 6 to discuss the fragmentation problem in fine-grained deduplicated data. Fig. 6(a) lists three backup streams, and Fig. 6(b) suggests the order-based data layout after fine-grained deduplicating these three backup streams. The order-based data layout allocates chunks in containers according to their written order and is widely used in previous works [12,23,37,52]. When restoring a backup, the needed and unneeded chunks are always mixed in this data layout. Consider Container 2 in Fig. 6(b) for example: when restoring the 3^{rd} backup, chunk F is needed while chunks D and E are unneeded, but all of them will be read as a whole container due to container I/O, which causes extra I/O overheads.

Rewriting-like defragmentation approaches could be extended to fine-grained deduplication to alleviate the fragmentation problem [53]. Their mechanisms can be summarised as skipping deduplicating chunks already in sparse containers, but this cannot stop the locality of deduplicated data becoming increasingly poorer as the number of backups increases, which thus makes the restore speed continually decrease [13,23,53].

MFDedup [58] introduces a lifecycle-based data layout and eliminates the fragmentation problem in chunk-level deduplication. The lifecycle-based data layout classifies chunks into categories according to whether they are always referenced by the same set of consecutive backup workloads (i.e., **lifecycles**), and stores chunks in the same category together. Lifecycle-based classification of chunks ensures whichever backup workload is to be restored, chunks in any categories are always either all needed together or all not needed together. Thus, reading needed chunks in the unit of categories will never cause unneeded chunks to be read. Generally, MFDedup only considers **one-level** simple reference relationships (between chunks and backup workloads), which is the only type of reference relationship in chunk-level deduplication.

However, directly applying this lifecycle-based data layout to fine-grained deduplication is not feasible since fine-grained deduplication introduces an additional kind of reference relationship between delta and base chunks and causes new fragmentation. In the 2^{nd} backup stream, there are **two-level** reference relationships:

- Between workloads and chunks.
  i.e., the 2^{nd} backup stream $\Leftrightarrow$ {A, B', C', H, I, F', G}
- Between base chunks and delta chunks.
  i.e., B $\Leftrightarrow$ Δ(B); C $\Leftrightarrow$ Δ(C); F $\Leftrightarrow$ Δ(F)

Therefore, we need a new data layout that considers both kinds of reference relationships to eliminate the fragmentation problem in fine-grained deduplicated data.

We first need a new way to describe chunks' lifecycles with the additional introduced reference relationship's impacts. Here we define the **Necessary Chunks** (denoted by **NC**) of a backup workload as the combination of its directly referenced chunks (i.e., the 1^{st} level) and its indirectly referenced chunks (i.e., the 2^{nd} level). Accordingly, we redefine a chunk's lifecycle in fine-grained deduplication as which backup workloads' NCs refer to this chunk, which could cover the two-level reference relationships. In Fig. 6, we can list the NCs for the three backups:

- NC_Backup1: A, B, C, D, E, F, G
- NC_Backup2: A, B, Δ(B'), C, Δ(C'), H, I, F, Δ(F'), G
- NC_Backup3: A, J, C, H, Δ(H'), I, F, Δ(F'), G, Δ(G')

In this example, the lifecycle of chunk *G* is from *NC_Backup1* to *NC_Backup3*, since *G* is used as a unique chunk for *NC_Backup1* & *NC_Backup2* and then as a base for *NC_Backup3*.

After that, we could build a **delta-friendly data layout** by integrating the second level of reference relationship into the lifecycle management as well. As shown in Fig. 6(c), the delta-friendly data layout consists of categories, which includes several chunks. To clearly present them, we use **Cat.(X,Y)** to indicate the category, which includes all chunks whose lifecycles are only from *NC_BackupX* to *NC_BackupY*. All deduplicated data are classified and sequentially stored in categories according to their lifecycles, which hugely benefits the restore workflow. In this example, *NC_Backup1* is composed of *Cat.(1,1)*, *Cat.(1,2)* and *Cat.(1,3)*; *NC_Backup2* is composed of *Cat.(1,2)*, *Cat.(2,2)*, *Cat.(1,3)* and *Cat.(2,3)*; *NC_Backup3* is composed of *Cat.(1,3)*, *Cat.(2,3)* and *Cat.(3,3)*. When restoring any of these three backups, we can select categories according to the above lists, and all chunks in selected categories are all needed. In this way, the restore workflow never needs to read any unneeded chunks, and the *Fragmentation Issue* in Challenge 2 could be eliminated.

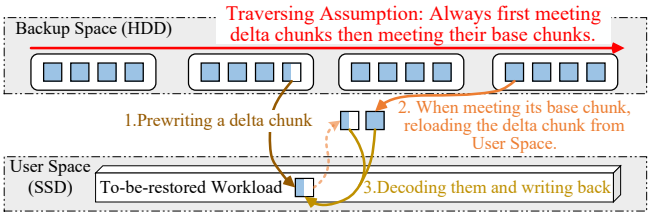


Figure 7: The delta prewriting mechanism. Here the half shaded chunk is a delta chunk.

To simplify the implementation of the delta-friendly data layout, we only deduplicate redundancies between adjacent backups to ensure that chunks' lifecycles are always consecutive (composed of successive backup streams' *Necessary Chunks*), similar to the approach in MFDedup [58]. This strategy may reduce the total deduplication ratio, but it will not be significant according to several previous works [38, 44, 58], which will be also further studied in §5.4.

3.4 Forward Reference and Delta Prewriting

For Challenge 3, we design a delta prewriting mechanism. It relies on two things: ① The storage media's I/O characteristics between User Space and Backup Space are asymmetric. Backup Space usually uses HDDs as storage media due to its lower price, while User Space usually uses SSDs or NVMs since better I/O performance is essential for business [58]. ② When performing a restore, delta-encoded chunks are always accessed before their base chunks, which we call "**Forward Reference.**"

Fig. 7 shows the basic idea of the delta prewriting mechanism. For each delta chunk, the prewriting mechanism will prewrite it to the offset where it should be after delta decoding in the to-be-restored backup workload (in User Space). And then, when meeting its base chunk later, the prewriting mechanism will read the delta chunk from the prewritten position, decode the delta chunk with the base, and finally write back the decoded chunk to its offset. Through this mechanism, we ensure that when restoring, all restore-involved containers only need to be read only once, which hugely reduces the I/O overheads on Backup Space.

The next issue is how to make the assumption always hold. By studying the data layout proposed in §3.3, we find it is possible to design a special path for traversing restore-involved containers when restoring, in which delta chunks always appear in front of their base chunks. We call it **"Always-Forward-Reference" traversing** (shortened to

AFR traversing), whose details will be introduced in §4.4.

Due to improved spatial locality (delta-friendly data layout in §3.3) and temporal locality (the AFR traversing and delta prewriting in §3.4) in deduplicated data, the I/O overheads in the restore workflow are hugely reduced. Meanwhile, there exists only sequential I/O to the Backup Space when restoring, which is optimized for HDDs. Finally, the restore speed could be greatly improved, which we evaluate in §5.3.

4 Design and Implementation

4.1 General Description

The overall framework of MeGA is shown in Fig. 8. In general, ① For the backup workflow, MeGA first runs *Chunk-level Deduplication* to remove duplicate chunks according to *Local-based FP Index*, and then, it finds similar matches for unique chunks according to *Local-based Sketch Index* and selectively applies delta compression using *Delta Selector*. ② For the storage organization, MeGA stores and manages the deduplicated and delta compressed data in the *Delta-Friendly Data Layout*. ③ For the restore workflow, MeGA generates an *Offset Hash Table* according to the recipe of a to-be-restored workload; then, MeGA accesses all restore-involved containers with *AFR Traversing* and *Delta Prewriting*.

Specifically, there are several modules in MeGA:

- *Local-based FP Index and Local-based Sketch Index* maintain fingerprints and sketches of each backup workload's chunks in separate hash tables per backup. They only retain the current and last backup's tables (similar to some previous works [44, 58]), because MeGA only deduplicates a backup within itself and the previous backup (mentioned in §3.3).
- *Chunk-level Deduplication* first splits the backup stream into chunks with Content-Defined Chunking [31, 46] and then calculates a fingerprint (i.e., SHA1 digest) for each chunk. After that, it detects and eliminates identical chunks with a Local-based FP index.
- *Delta Selector* first generates sketches with the resemblance detection approaches [4, 5, 37, 52, 57] for unique chunks and identifies similar candidates according to the Local-based Sketch index for further delta compression. Then, it delta-encodes chunks unless the referenced bases are in base-sparse containers.
- *Base Cache* holds cached containers to provide base chunks for delta compression in the backup workflow.
- *Delta-Friendly data layout* manages fine-grained deduplicated chunks according to their lifecycles, reflecting which backup workloads require these chunks. As a result, the delta-friendly data layout promises to eliminate the fragmentation problem in fine-grained deduplicated data and reduce I/O overheads in the restore workflow.
- *AFR Traversing* applies "Always-Forward-Reference" traversing on fine-grained deduplicated data in a restore workflow, which guarantees that delta chunks are always accessed before their base chunks and provides the pre-condition for Delta Prewriting.
- *Delta Prewriting* transfers the random operations from Backup Space to User Space, exploiting the asymmetry of storage media characteristics between the two spaces. This also avoids repeatedly accessing containers when restoring files.
- *Offset Hash Table* is built according to a to-be-restored backup workload's recipe and provides offsets of chunks (three kinds of chunks: unique, base, and delta) in the to-be-restored backup workload.

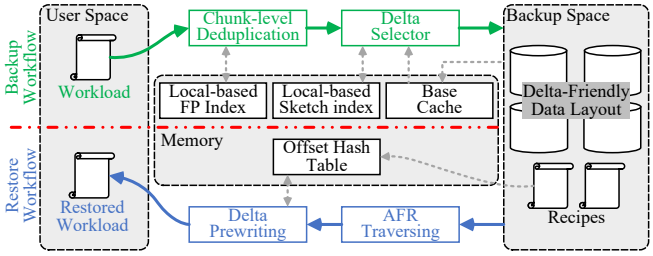


Figure 8: An overview of MeGA framework.

Details of each workflow using our proposed key techniques will be introduced in the following §4.2–§4.4.

4.2 Backup Workflow

The backup workflow runs Chunk-level Deduplication and Delta Selector to eliminate duplicate chunks and redundancies among similar chunks, respectively.

Chunk-level Deduplication. The chunk-level deduplication step splits the backup stream into chunks with Content-Defined Chunking [31, 46] and then calculates a fingerprint (i.e., SHA1 digest) for each chunk. After that, MeGA detects and removes duplicate chunks according to the Local-based FP Index, as we introduced in §4.1.

Delta Selector. Then, the backup workflow runs Delta Selector with the following steps. ① Delta Selector first combines several successive chunks (from Chunk-level Deduplication) into fix-sized segments (e.g., 20MB). ② In each segment, Delta Selector generates sketches (i.e., Super Features [22]) for each (unique) chunk, and then tries to find for each chunk a similar chunk as its base chunk with the Local-based Sketch Index. ③ For chunks that have a potential base chunk, Delta Selector records their base chunk's container ID in a 'selector table', which counts the times each container is referenced for base chunks within a segment. ④ Then, Delta Selector observes which containers are rarely referenced (with a **threshold**) in the 'selector table' and considers these containers as 'sparse-base containers', which are inefficient to read for base chunks. ⑤ Finally, for chunks having a similar chunk that is not in sparse-base containers, Delta Selector will run delta compression to calculate and store their differences (i.e., delta chunk) for saving space; For the remaining chunks, they will be directly stored as unique chunks. Base chunks in delta compression are acquired from the base cache, and if a cache miss occurs, the base cache will read related containers

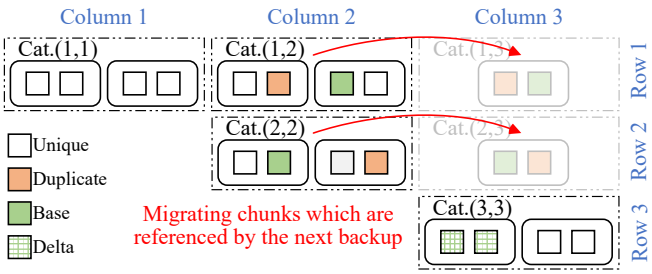


Figure 9: An example of maintaining data layout after storing the 3^{rd} *backup*. Chunks, which are duplicate to the 3rd backup and referenced as base chunks in the 3rd backup, will be migrated to new categories. *Cat(1,3)* and *Cat.(2,3)* do not exist before migrations.

from disks and add them to the cache.

As a result, Delta Selector could improve the efficiency of reading base chunks and then accelerate the backup workflow. Next, we will introduce how to store these deduplicated data.

4.3 Maintaining Delta-Friendly Data Layout

In this subsection, we will introduce how to locate the deduplicated data in the delta-friendly data layout. There are two steps: ① Store the incoming deduplicated data of a new backup in the delta-friendly data layout. ② Process the incoming and previous backups' deduplicated data to ensure each chunk's location is consistent with the principle of our delta-friendly data layout.

Storing New Fine-grained Deduplicated Data and Data Organization. For storing fine-grained deduplicated data, we first consider their lifecycles. After running the backup workflow (introduced in §4.2), the fine-grained deduplicated data consists of the latest backup workload's unique and delta chunks. Since these chunks are only referenced by the latest backup, they should have the same lifecycle, and their lifecycle should be different from previously stored chunks.

Then, considering the definition of the lifecycle and the naming style of categories (**shorten to Cat.**) introduced in §3.3, these chunks (assuming they are from the n^{th} Backup) should be classified into a new category *Cat.(n,n)*.

Considering the sizes of categories are usually variable, we design a two-level storage organization: fix-sized Containers (e.g., 4MB) and variable-sized Categories. Containers directly hold chunks, and categories hold containers whose chunks have the same lifecycle. For example, *Cat.(1,2)* could include one or several containers, and each container holds chunks whose lifecycle is from *NC_Backup1* to *NC_Backup2*.

Data Migration. After storing fine-grained deduplicated data of the latest backup workload, we should consider updating the data layout to handle the issue that some chunks' lifecycles are changed. In general, storing a new backup in the delta-friendly data layout only changes the lifecycles of its adjacent backups' chunks, because MeGA only allows adjacent backups to share common chunks (i.e., MeGA deduplicates a backup within itself and its previous backup). Therefore, these shared chunks' lifecycles should be extended to the latest backup. Thus, we need to migrate these shared chunks into new categories to match their updated lifecycles, and we call these migrations the maintenance workflow.

An example of the maintenance workflow is shown in Fig. 9. It shows a situation that the 1^{st} and 2^{nd} backups have been stored in the data layout, and the 3^{rd} backup is the latest one, whose fine-grained deduplicated data have been stored in *Cat.(3,3)*, as discussed earlier in this subsection. At this time, some chunks located in *Cat.(1,2)* and *Cat.(2,2)* are referenced by the 3rd backup (as duplicate or base chunks). Thus, these chunks' lifecycles newly include *NC_Backup3* and they should be migrated into new categories. In this example, chunks in *Cat.(1,2)* and *Cat.(2,2)* will be traversed, and duplicate/base chunks will be migrated into new categories *Cat.(1,3)* and *Cat.(2,3)*, respectively.

Note that the **maintenance workflow (i.e., data migration) only works on related categories and does not involve all categories.**. As the example in Fig. 9 shows, a maintenance workflow after storing the 3^{rd} backup only impacts Column 2. Similarly, the maintenance workflow always runs on one column, and its overhead is also limited (will be studied in §5.6). With support of the maintenance workflow, the delta-friendly data layout is preserved, which benefits restore performance.

Features in Migration. There exist two interesting features when the maintenance workflow involves delta and base chunks. For clarity, here we say Cat.(X,Y) is in Row X, and Column Y, as shown in Fig. 9.

Feature 1: base chunks are always in the same or an earlier Row than their delta chunks. It could be easily explained by the example in Fig. 9. For delta chunks of the 3^{rd} backup (must be in *Cat.(3,3)*), their base chunks can only be from two sources: ① from the 3^{rd} backup itself. In this case, the bases are also in *Cat.(3,3)*, the same Row as the delta. ② from the 2^{nd} backup. In this case, the bases must be migrated into *Cat.(1,3)* or *Cat.(2,3)*, the earlier Row than the delta.

Feature 2: base chunks are always in the same or a later Column than their delta chunks. Here we also take Fig. 9 as an example: When the duplicate chunks in Fig. 9 contain base or delta chunks, there are two cases: ①If a delta chunk is a duplicate of another delta chunk (i.e., its "original" chunk is a duplicate of delta-encoded chunk) to the 3rd backup and should be migrated, its base must be also migrated as the delta's dependency, since both of them should be included in NC_Backup3. Therefore, in this case, they will be migrated into the same Column (Column 3). ②If a base chunk is duplicate (i.e., is itself a duplicate) to the 3rd backup and should be migrated, its delta will not be migrated, since the 3rd backup does not require this delta chunk. In this case, the base (migrated to Column 3) will be in a later Column than the delta (left in Column 2).

These two features will help to achieve "Always-Forward-Reference" traversing, which will be further used in §4.4.

Table 1: Possible category locations of the corresponding base chunks for the delta chunks in the 2^{nd} backup.

Delta Chunks' Positions		Corresponding Base Chunks' Possible Positions
Cat.(1,2)	⇒	Cat.(1,2), Cat.(1,3)
Cat.(2,2)	⇒	Cat.(1,2), Cat.(2,2), Cat.(1,3), Cat.(2,3)
Cat.(1,3)	⇒	Cat.(1,3)
Cat.(2,3)	⇒	Cat.(1,3), Cat.(2,3)

4.4 Restore Workflow

As introduced in §4.1, the restore workflow of MeGA relies on AFR traversing and Delta Prewriting. In the beginning, the restore workflow needs to determine which containers are needed for restoring the required backup workload.

Identifying All Required Containers. All the required containers could be simply calculated in a delta-friendly data layout. For example, there are n backup workloads stored, and we want to restore a backup B_k. According to the naming style of categories (mentioned in §3.3), all categories whose lifecycles include *NC_Backupk* are required, and they are $\cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j), where\ 1 \le i \le k \le j \le n$. For example, when restoring the 2^{nd} backup in Fig. 9, *Cat.(1,2)*, *Cat.(2,2)*, *Cat.(1,3)*, *Cat.(2,3)* are required.

Then all containers in these categories are the restore-required ones. Benefiting from the delta-friendly data layout, all chunks in these containers are exactly what we need, which avoids reading unneeded chunks when restoring. Next, we present how to traverse them for restoring a workload.

AFR Traversing. As mentioned in §4.1, AFR traversing promises that when traversing the restore-involved containers, delta chunks always appear in front of their base chunks. For the example in Fig. 6(c), when restoring the 2^{nd} backup, restore-involved categories are *Cat.(1,2)*, *Cat.(2,2)*, *Cat.(1,3)* and *Cat.(2,3)* (according to "Identifying All Required Containers"). In this case, we can achieve AFR traversing with the following order: *Cat.(2,2)* ⇒ *Cat.(1,2)* ⇒ *Cat.(2,3)* ⇒ *Cat.(1,3)*, in which we always meet the delta chunks before their base chunks.

Next, we explore how and why MeGA can achieve AFR traversing, also with the example of restoring the 2^{nd} backup in Fig. 6(c). Consider the two key Features about the relative positional relationship (i.e., the located categories' Rows and Columns) between the delta and base chunks (learned from §4.3). We can get Table 1, listing all possible positions (i.e., located categories) of delta and based chunks of the 2^{nd} backup. To achieve AFR traversing (accessing delta chunks and then their bases), *Cat.(2,2)* must be first accessed, which is because the base chunks of *Cat.(2,2)*'s delta chunks could be in all four categories as shown in Table 1. With similar analysis, we could finally get the previous example path: *Cat.(2,2)* ⇒ *Cat.(1,2)* ⇒ *Cat.(2,3)* ⇒ *Cat.(1,3)*. Additionally, AFR traversing should go through chunks and also containers of each category in reverse order in case there are delta and base chunks in the same category or container, since the delta must be generated and then appear after its base in the backup workflow.

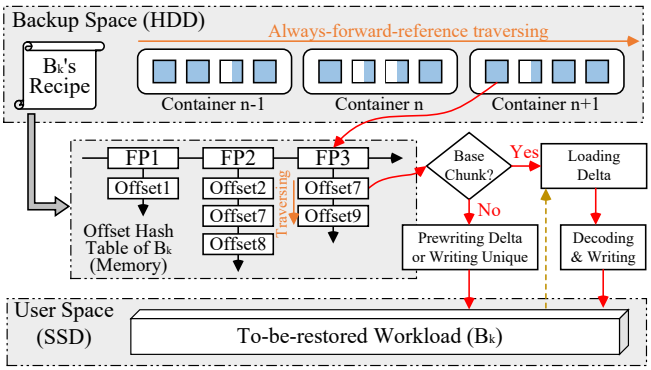


Figure 10: An example of the restore workflow.

To this end, we can summarize **three rules to achieve AFR traversing** on our delta-friendly data layout in general cases:

- Between columns, access columns in positive order. This is deduced from Feature 2 (in Section 4.3).
- In the same column, access categories in reverse order. This follows from Feature 1 (in Section 4.3).
- In a category, access containers in each category and chunks in each container in reverse order. This is because delta chunks can only reference earlier chunks by design.

Delta Prewriting. As shown in Fig. 10, *Delta Prewriting* requires an *Offset Hash Table*, which is generated according to the to-be-restored backup's recipe. The Offset Hash Table records key/value pairs: **each chunk's offset** (in the to-be-restored backup) and **whether it is a base chunk** (i.e., <offset, base tag>). For unique chunks in the recipe, we only insert its offset into its FP's entry list and tag this record as not a base (e.g., <offsetUniqueK, false>). For a delta chunk in the recipe, we first process it as a unique chunk (e.g., insert <offsetDeltaN, false> in its FP's entry list) and then additionally insert a record into the entry list of its base's FP and tag this record as a base chunk (e.g., <offsetDeltaN, true>).

Then, we apply AFR traversing on restore-involved containers. For each chunk, we acquire its entry list according to its fingerprint. We check each record in the entry list: If it is not a base chunk record, we directly write the chunk (it may be a delta or unique chunk) to the offset in the record; If it is a base chunk, we read the delta chunk from the offset in the record (the delta should already be written before), decode the delta chunk with the base chunk, and then write back the decoded chunk to the offset in the record.

Finally, MeGA could achieve a much higher restore speed with the benefits of the delta-friendly data layout, AFR traversing and delta prewriting, since it no longer reads unneeded chunks and repeatedly accesses restore-involved containers.

4.5 Discussion

In this subsection, we discuss several features and issues.

Deletion. Different from the order-based data layout, the delta-friendly data layout supports direct deletions without GC. Because MeGA allows workloads to share chunks as duplicate chunks or base chunks, deleting the n^{th} backup only needs to remove its unique chunks. According to the category naming rule, *Cat.(n,n)* only contains chunks unique to backup *n*. Thus, deleting the n^{th} backup could be achieved by directly removing this category, instead of the way in the order-based data layout, which first runs logical deletion and later runs garbage collection to reclaim storage space [3, 9, 14].

Delta Prewriting. This mechanism introduces additional I/O on User Space, including prewriting and reading delta chunks. Our observations suggest these issues cause about 5%-10% additional I/O overheads on User Space. Moreover, since delta chunks are usually much smaller than unique ones, we could also introduce a delta cache in memory and prewrite delta chunks into the cache as an alternative solution.

Memory Overhead. Since the size of the base chunk cache (in a backup workflow) is configurable, the other memory overhead of MeGA is mainly from the local-based indexes (in backup workflow) and the offset hash table (in restore workflow). ① Instead of putting the whole index in memory, MeGA only maintains the index of the last two backups and thus costs less memory, which is similar to some previous work [44, 58]. Moreover, a stream-informed index [56] could also be applied to our local-based index to further reduce memory overhead. ② The overhead of the offset hash table is related to the number of chunks in a single backup. Some previous works [1, 43] suggest that the majority of single backups were 4–128GB, and for these cases, RAM usage for the Offset Hash Table could be 12.9–445.6MB, which is feasible for a server. To reduce this RAM usage for large backups, we could keep the offset hash table in an on-disk key-value store, but it would require indexing time.

Maintenance's (i.e., Migrations) Overheads. The Maintenance process in MeGA replaces Garbage Collection (GC) in previous works, and its overhead could be offset since both techniques are offline processes, which will be evaluated in §5.6. Through the Maintenance process, MeGA could achieve direct deletion (to immediately reclaim storage space) instead of logical deletion followed by GC. Besides, the Maintenance process also addresses two interesting issues [36]: knowing how much space will be freed after deletion and estimating the remaining logical space of a fine-grained deduplication system.

Incremental Backups. Although MeGA focuses on full backups (i.e., a full snapshot of primary storage), we present a plan to support incremental backups by generating "virtual" full backups.

When handling an incremental backups, we generate a "virtual" full backup according to the previous full backup's recipe, and then process the "differences" included in the incremental backup. Specifically, ① non-modified (i.e., not listed in the incremental backup) parts of the "virtual" full backup are duplicates, and we can directly copy corresponding records from the previous full backup's recipe to the "virtual" full backup's recipe; ② modified (i.e., listed in the incremental backup) parts have potentially new content, and we need to apply fine-grained deduplication. Chunk boundaries need to be recalculated due to the modified data regions, so we could combine the new data with their surrounding duplicate chunks to make up a local stream and run content-defined chunking on this stream to determine new chunks. Then, MeGA could process these chunks normally, and finally record these "modifications" in the "virtual" full backup's recipe.

5 Evaluation

5.1 Configuration

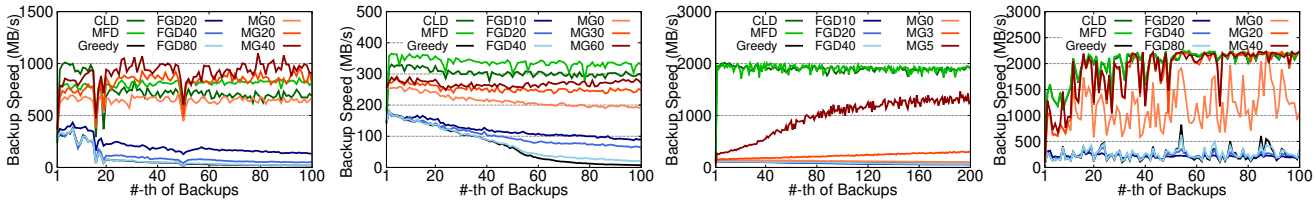
We perform our experiments on a workstation running Ubuntu 18.04 with an Intel Core i7-8700 @ 3.2GHz CPU, 64GB memory, and a 7200rpm HDD. To better evaluate MeGA, the following five approaches are considered:

- **Greedy**: applying the greedy strategy for fine-grained deduplication, often evaluated as the baseline [44, 53].
- **FGD**: Fine-Grained Deduplication with the Capping rewriting technique [23], which skips some deduplication and delta compression whose duplicate chunks or base chunks are located in a few referenced containers. This is similar to a recent work called SDC [53].
- **CLD**: Chunk-Level Deduplication with the Capping rewriting technique [23], considered as a typical approach of chunk-level deduplication defragmentation.
- **MFD**: Chunk-level deduplication with the previous lifecycle-based data layout, which only deduplicates chunks between adjacent backups [58].

These approaches are implemented according to related papers, and they all follow these common configurations:

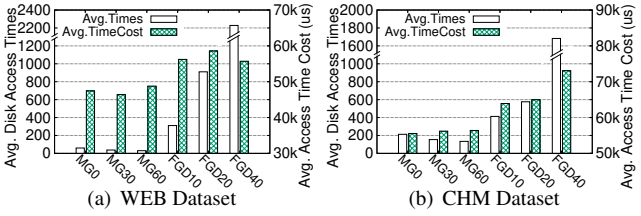
- Chunking backups uses FastCDC [46] with the minimum, average, and maximum chunk sizes set to 2KB, 8KB, and 64KB; SHA1 is used for chunk identification.
- Their resemblance detection generates 12 features and 3 super-features as sketches for each unique chunk, as suggested in previous works [22, 24, 37].
- Consecutive chunks are compressed together with ZSTD and stored in containers.
- The delta encoding stage uses Xdelta to calculate differences between unique chunks and its similar candidates, as configuration in previous works [52, 57].
- MeGA only requires a container cache in the backup workflow, and the other four non-trivial approaches require two container caches in both backup and restore workflows. The cache of each workflow totals 512MB. When loading base chunks into the cache, all approaches apply container I/O for fair comparison.

To focus on testing the performance of the deduplication storage side (i.e., running deduplication on HDD media), tested datasets are backed up from User Space (i.e., a



(a) WEB Dataset (b) CHM Dataset (c) SYN Dataset (d) VMS Dataset

Figure 11: Backup speed of five deduplication approaches on four datasets.



(a) WEB Dataset (b) CHM Dataset

Figure 12: Disk access times and time cost in MeGA and FGD. We only show WEB and CHM due to the space limit.

Table 2: Four backup datasets used in evaluation.

Name	Original Size	Versions	Workload Descriptions
WEB	269 GB	100	Backups of website: news.sina.com, captured from Jun. to Sep. in 2016.
CHM	279 GB	100	Source codes of Chromium project from v82.0.4066 to v85.0.4165
SYN	1.38 TB	200	Synthetic backups by simulating file create/delete/modify operations [42]
VMS	1.55 TB	100	Backups of an Ubuntu 12.04 Virtual Machine

RamDisk) to Backup Space (i.e., a 7200rpm HDD) one by one while the restore runs in the reverse direction. For speed of backup and restore in our evaluation, we present the average results of five runs.

Four backup datasets are used for evaluation, as shown in Table 2. These datasets represent various typical backup workloads, including website snapshots, an open-source code project, virtual machine images, and a synthetic dataset. They have been used in several deduplication studies [8, 46, 53].

5.2 Backup Speed

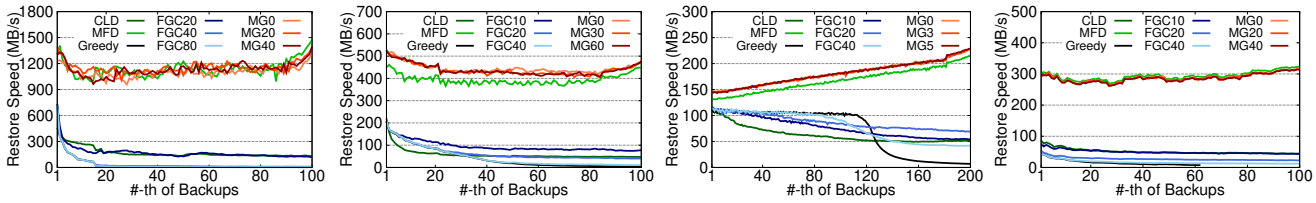
The backup speed of five approaches are evaluated and shown in Fig. 11, and the results vary by 5.2% on average in multiple runs. FGD# and MG# represent FGD and MeGA with different parameters (the capping level in FGD and the delta selector threshold in MeGA). The capping level L indicates that when processing a backup stream segment, containers will be considered as sparse containers except for the L most referenced (for duplicate or base chunks). Chunks in the segment, whose duplicate or base chunks are in sparse containers, will be processed as unique chunks. A delta selector threshold T means that when processing a backup stream segment, containers, which are referenced for base chunks less than T times, will be considered as base-sparse containers. Delta compression in the segment, whose base chunks are in base-sparse containers, will be skipped. Considering that datasets have different characteristics and require different parameters, we optimized the parameters for each dataset. The backup speed is calculated by $\frac{The-Size-of-Backup-or-Restore-Workload}{Backup-or-Restore-Time-Cost}$. Because deduplicated and compressed data is much smaller than their original size and writing to disk takes less time, the backup speed could exceed the disk speed.

With the benefits of the delta selector, MeGA outperforms other fine-grained deduplication approaches (i.e., Greedy and FGD). On the SYN dataset, MeGA reads increasingly fewer containers when processing later backups, which makes MeGA's backup speed increase. VMS is a virtual machine dataset and its modification style (i.e., trending to change the same region in each backup) makes distribution of base chunks uneven, which makes MeGA0's performance jittery. Generally, MeGA achieves a 4.47–34.45× higher backup speed than Greedy.

Fig. 11 also suggests a stricter (smaller) capping level in FGD and a stricter (bigger) delta selector threshold in MeGA both accelerate backup speed, due to skipping some potential delta compression and the need to read more base chunks. Note that if delta selector threshold and capping level were strict enough, all delta compression would be skipped. Though the delta selector and the capping rewriting have similar mechanisms, their results are much different due to their different views on container utilization. The capping rewriting is restore-workflow-oriented and focuses on how many needed chunks (all kinds of chunks) are in containers. But the delta selector is backup-workflow-oriented, and only concerns how many base chunks are in containers.

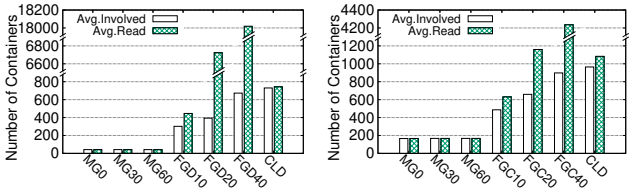
Fig. 12 further studies why MeGA could achieve a much higher backup speed than FGD, which lists the disk access times for acquiring base chunks (within the unit of containers) and average access time cost when storing backup. On the one hand, MeGA has much lower disk access time due to skipping reading "inefficient" containers. On the other hand, MeGA has a lower average access time cost, since it only finds base chunks in adjacent backups and its accessed containers will be located closer. These two efforts ensures MeGA's better performance.

Note that MeGA achieves similar results with chunk-level deduplication approaches (CLD and MFD) on most datasets. It is because the additional I/O and computation overhead both have been hugely limited by our delta selector and previous computation optimization works, respectively. Besides, SYN is a synthetic dataset and its modified parts are distributed ran-



(a) WEB Dataset (b) CHM Dataset (c) SYN Dataset (d) VMS Dataset

Figure 13: Restore speed of five deduplication approaches on four datasets.



(a) WEB Dataset (b) CHM Dataset

Figure 14: Number of restore-involved containers and actually read containers in MeGA, FGD and CLD. Only WEB and CHM are shown due to the space limit.

domly instead of having more typical locality, which makes MeGA slightly slower than CLD and MFD.

5.3 Restore Speed

Fig. 13 shows the restore speed of all five approaches, and the results vary by 2.6% on average in multiple runs. Among all approaches, MeGA consistently achieves a better restore performance than other approaches, which reflects MeGA's restore techniques (i.e., the delta-friendly data layout, delta prewriting and AFR traversing). MeGA solves the *Fragmentation Issue* and *Repeatedly Accessing Issue* and improves the spatial and temporal locality in fine-grained deduplicated data. It also ensures that MeGA's restore performance is more consistent, while FGD, CLD and Greedy all have a decreasing restore speed. Note that the local compression ratio increases when storing more backups on SYN, which makes the restore speed faster. Generally, MeGA achieves a 30–105× higher restore speed than Greedy.

Fig. 14 shows the number of restore-involved containers (i.e., containers with restore-required chunks) and containers read from disk during restore of MeGA, FGD, and CLD. These two metrics reflect the seriousness of the *Fragmentation Issue* and *Repeatedly Accessing Issue*, respectively. Compared with FGD and CLD, MeGA has lower results on both of the metrics due to applying our data layout and AFT traversing with delta prewriting. Consequently, MeGA achieves a much higher restore performance, as shown in Fig. 13.

5.4 Deduplication Ratio

Fig. 15 studies deduplication ratios of five approaches with different parameters mentioned in the above subsections. All three fine-grained deduplication approaches (i.e., Greedy, FGD, and MeGA) have higher deduplication ratios than chunk-level deduplication approaches (i.e., CLD and MFD), since they can exploit compressibillity among similar chunks. Greedy always achieves the highest deduplication ratio, and MeGA achieves similar results. For FGD and MeGA, a stricter capping level in FGD or threshold in MeGA will lower the deduplication ratio but lead to a better backup or restore speed, as reported in the above subsections.

MeGA's advantage is relatively smaller on the VMS and SYN datasets. For VMS, its modification style (i.e., trending to change the same region in each backup) leads to fewer similar chunks, which limits the benefits of fine-grained deduplication, regardless of the approach. For SYN, its modifications are completely random because it is a synthetic dataset, and the locality of base chunks is not as strong as that of other datasets. Therefore, MeGA's delta selector causes more reduction in the compression ratio.

Generally, MeGA preserves fine-grained deduplication's significant advantage by achieving a 1.18–8.73× higher deduplication ratio than chunk-level approaches.

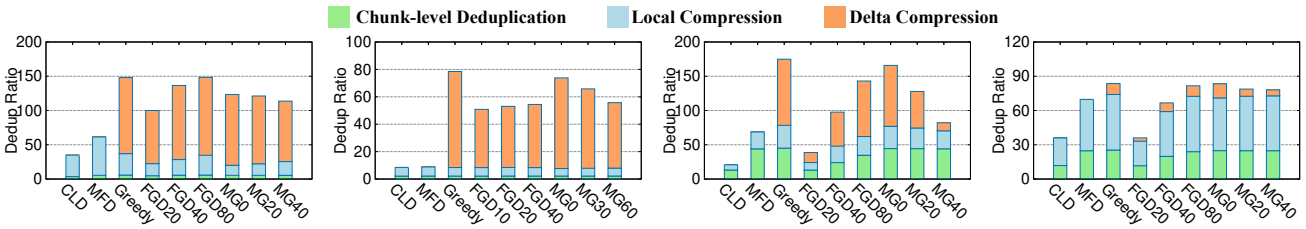
5.5 Overall Performance

The three metrics discussed above are of the most interest to users. Fig. 16 shows the overall performance with different parameters (used in Fig. 11 and 13) from the above section. It is obvious that MeGA significantly improves over other fine-grained deduplication approaches (i.e., FGD and Greedy) on both backup and restore speed while preserving the deduplication ratio advantage of fine-grained deduplication. It reflects the performance improvement that our proposed technology brings. MeGA's advantage is relatively smaller on SYN and VMS datasets. As we mentioned in § 5.4, it is because VMS does not have many similar chunks, and SYN lacks natural locality, which is unfriendly for our delta selector.

5.6 I/O Overhead in Maintaining Data Layout

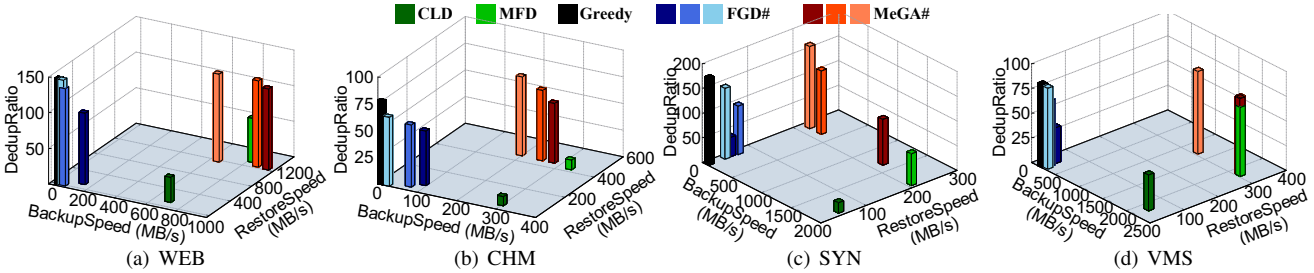
In this subsection, we evaluate I/O overheads of maintaining the delta-friendly data layout (shortened to "Maintenance") compared with traditional garbage collection (GC).

Our experiments are based on MeGA and FGD using the median parameters as in Fig. 11 and 13, and MeGA runs maintenance while FGD runs GC. For GC, a container liveness threshold is usually considered to make a tradeoff between more storage space cost and more GC overheads. Here we use three liveness thresholds: 0%, 25%, and 50%, mapping to toleration up to 0%, 25%, 50% invalid chunks in each container, respectively. In order to make the results among different datasets comparable, we measure their time costs. Both approaches retain the last 20 backups; thus GC would



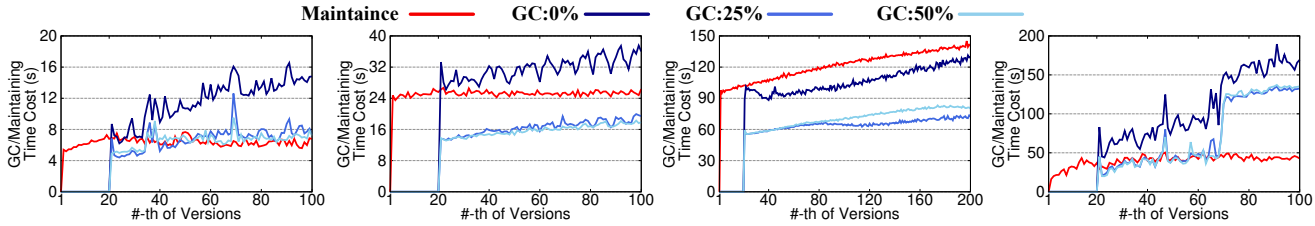
(a) WEB Dataset (b) CHM Dataset (c) SYN Dataset (d) VMS Dataset

Figure 15: Deduplication ratio achieved by the ten approaches on four datasets.



(a) WEB (b) CHM (c) SYN (d) VMS

Figure 16: A general view of five approaches. MeGA and FGD use three different parameters as used in Fig. 11, 13 and 15. The results of backup and restore speed are from the average performance on each dataset's last 10 backups.



(a) WEB Dataset (b) CHM Dataset (c) SYN Dataset (d) VMS Dataset

Figure 17: The delta-friendly data layout's maintenance vs. the order-based data layout's garbage collection.

not run for the first 20 backups, though the maintenance does.

Fig. 17 compares time cost of maintenance and GC. For GC, a bigger threshold does not always lead to a lower I/O overhead, since tolerating invalid chunks will make the next GC need to clean more containers, which causes additional I/Os. In general, maintenance and GC have similar I/O overheads, and compared with the best version of GC ("GC:25%"), maintenance costs about 0.32–1.92× the GC I/O overheads, which suggests maintenance and GC's overhead have different characteristics and have an overall similar impact.

Note that in maintenance of MeGA, if all chunks in a container are needed to be migrated to a new category, we can directly let this container belong to that new category without any chunk migration. It is interesting to observe that about 25.13% (WEB), 14.57% (CHM), 59.13% (SYN), and 72.95% (VMS) of containers do not need chunk migrations.

6 Conclusion

This paper proposes MeGA, a fine-grained deduplication framework, with three techniques: backup-workflow-oriented delta selector, delta-friendly data layout, and AFR traversing with delta prewriting, to address the three issues for different forms of poor locality caused by the introduction of delta compression: reading base chunks, fragmentation, and repeatedly accessing containers, respectively. Evaluations show that MeGA achieves performance close to chunk-level deduplication while preserving fine-grained deduplication's significant deduplication ratio advantage.

Acknowledgments

We are grateful to our shepherd and the anonymous reviewers for their insightful comments. This work was supported in part by NSFC under Grant 61972441, 61972112 and 61832004, in part by Shenzhen Science and Technology Program under Grants RCYX20210609104510007, JCYJ20210324131203009, JCYJ20200109113427092, and GXWD20201230155427003-20200821172511002, in part by Guangdong Basic and Applied Basic Research Foundation under Grant 2021A1515012634, 2021B1515020088 and in part by the HITSZ-J&A Joint Laboratory of Digital Design and Intelligent Fabrication under Grant no. HITSZ-J&A-2021A01.

Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proceedings of the 2015 USENIX Annual Technical Conference*, 2015.

[2] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[3] Fabiano C. Botelho, Philip Shilane, Nitin Garg, and Windsor Hsu. Memory efficient sanitization of a deduplicated storage system. In *Proceedings of the 11th USENIX conference on File and Storage Technologies*, pages 81–94, 2013.

[4] Andrei Z. Broder. On the resemblance and containment of documents. In *Proceedings of 1997 Compression and Complexity of SEQUENCES*, 1997.

[5] Andrei Z. Broder. Identifying and filtering near-duplicate documents. In *Proceedings of the 11th Combinatorial Pattern Matching Annual Symposium*, 2000.

[6] Zhichao Cao, Shiyong Liu, Fenggang Wu, Guohua Wang, Bingzhe Li, and David H. C. Du. 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*, 2019.

[7] Licheng Chen, Zhipeng Wei, Zehan Cui, Mingyu Chen, Haiyang Pan, and Yungang Bao. CMD: classification-based memory deduplication through page access characteristics. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.

[8] Liangfeng Cheng, Yuchong Hu, Zhaokang Ke, and Zhongjie Wu. Coupling right-provisioned cold storage data centers with deduplication. In *Proceedings of the the 50th International Conference on Parallel Processing*, 2021.

[9] Fred Douglis, Abhinav Duggal, Philip Shilane, Tony Wong, Shiqin Yan, and Fabiano C. Botelho. The logic of physical garbage collection in deduplicating storage. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies*, pages 29–44, 2017.

[10] Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 12th ACM SIGCOMM Internet Measurement Conference*, 2012.

[11] Abhinav Duggal, Fani Jenkins, Philip Shilane, Ramprasad Chinthekindi, Ritesh Shah, and Mahesh Kamat. Data domain cloud tier: Backup here, backup there, deduplicated everywhere! In *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.

[12] 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 Trans. Parallel Distributed Syst.*, 27(3):855–868, 2016.

[13] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proceedings of the 2014 USENIX Annual Technical Conference*, 2014.

[14] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of 2011 USENIX Annual Technical Conference*, pages 1–14, 2011.

[15] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[16] Navendu Jain, Michael Dahlin, and Renu Tewari. TAPER: tiered approach for eliminating redundancy in replica synchronization. In *Proceedings of the 3rd Conference on File and Storage Technologies*, 2005.

[17] Keren Jin and Ethan L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of of SYSTOR 2009: The Israeli Experimental Systems Conference*, 2009.

[18] Jonghwa Kim, Choonghyun Lee, Sang Yup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. Deduplication in ssds: Model and quantitative analysis. In *Proceedings of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, 2012.

[19] Keonwoo Kim, Jee-hong Kim, Changwoo Min, and Young Ik Eom. Content-based chunk placement scheme for decentralized deduplication on distributed file systems. In *Proceedings of the 13th International Conference on Computational Science and Its Applications*, 2013.

[20] Ricardo Koller and Raju Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, 2010.

[21] Lucas Kuhring and Zsolt István. Storing parquet tile by tile: Application-aware storage with deduplication. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications*, 2019.

[22] Purushottam Kulkarni, Fred Douglis, Jason D. LaVoie, and John M. Tracey. Redundancy elimination within large collections of files. In *Proceedings of the 2004 USENIX Annual Technical Conference*, 2004.

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

[24] Xing Lin, Guanlin Lu, Fred Douglis, Philip Shilane, and Grant Wallace. Migratory compression: coarse-grained data reordering to improve compressibility. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*, 2014.

[25] Jian Liu, Yunpeng Chai, Chang Yan, and Xin Wang. A delayed container organization approach to improve restore speed for deduplication systems. *IEEE Trans. Parallel Distributed Syst.*, 27(9):2477–2491, 2016.

[26] Joshua P. MacDonald. File system support for delta compression, 2000.

[27] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, 2016.

[28] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. *ACM Trans. Storage*, 7(4):14:1–14:20, 2012.

[29] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Trans. Computers*, 60(6):824–840, 2011.

[30] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for http. *SIGCOMM Comput. Commun. Rev.*, 27(4):181–194, 1997.

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

[32] Mohammad Nasirifar and Angela Demke Brown. Deduplicating future data transfer using data exchanged in the past to decrease mobile bandwidth usage. In *Proceedings of the 18th Annual International Conference on Mobile Systems, Applications, and Services*, 2020.

[33] Lars Nielsen, Dorian Burihabwa, Valerio Schiavoni, Pascal Felber, and Daniel E. Lucani. Minervafs: A user-space file system for generalised deduplication: (practical experience report). In *Proceedings of the 40th International Symposium on Reliable Distributed Systems*, 2021.

[34] Sungbo Park, Ingab Kang, Yaebin Moon, Jung Ho Ahn, and G. Edward Suh. BCD deduplication: effective memory compression using partial cache-line deduplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[35] Himabindu Pucha, David G. Andersen, and Michael Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation*, 2007.

[36] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *Proceedings of the 8th USENIX Workshop on Hot Topics in Storage and File Systems*, 2016.

[37] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu. Wan-optimized replication of backup datasets using stream-informed delta compression. *ACM Trans. Storage*, 8(4):13:1–13:26, 2012.

[38] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu. Delta compressed and deduplicated storage using stream-informed locality. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems*, 2012.

[39] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS*, 2008.

[40] Zhen Jason Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, and Erez Zadok. Cluster and single-node analysis of long-term deduplication patterns. *ACM Trans. Storage*, 14(2):13:1–13:27, 2018.

[41] Haoliang Tan, Zhiyuan Zhang, Xiangyu Zou, Qing Liao, and Wen Xia. Exploring the potential of fast delta encoding: Marching to a higher compression ratio. In *Proceedings of the 2020 IEEE International Conference on Cluster Computing*, 2020.

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

[43] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu. Characteristics of backup workloads in production systems. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*, 2012.

[44] 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.

[45] Avani Wildani, Ethan L. Miller, and Ohad Rodeh. HANDS: A heuristically arranged non-backup in-line deduplication system. In *Proceedings of the 29th IEEE International Conference on Data Engineering*, 2013.

[46] 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. *Proc. IEEE*, 104(9):1681–1710, 2016.

[47] Wen Xia, Hong Jiang, Dan Feng, and Lei Tian. DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads. *IEEE Trans. Computers*, 65(6):1692–1705, 2016.

[48] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Yukun Zhou. Ddelta: A deduplication-inspired fast delta compression approach. *Perform. Evaluation*, 79:258–272, 2014.

[49] 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 *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems*, 2015.

[50] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, and Gregory R. Ganger. Online deduplication for databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[51] Lianghong Xu, Andrew Pavlo, Sudipta Sengupta, Jin Li, and Gregory R. Ganger. Reducing replication bandwidth for distributed document databases. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, 2015.

[52] 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 *Proceedings of the 17th USENIX Conference on File and Storage Technologies*, 2019.

[53] 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 Trans. Parallel Distributed Syst.*, 31(10):2302–2314, 2020.

[54] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali Raza Butt. Duphunter: Flexible high-performance deduplication for docker registries. In *Proceedings of the 2020 USENIX Annual Technical Conference*, 2020.

[55] Feng Zhou, Li Zhuang, Ben Y. Zhao, Ling Huang, Anthony D. Joseph, and John Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proceedings of the 2003 ACM/IFIP/USENIX International Middleware Conference*, 2003.

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

[57] 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 *Proceedings of the 37th IEEE International Conference on Data Engineering*, 2021.

[58] Xiangyu Zou, Jingsong Yuan, Philip Shilane, Wen Xia, Haijun Zhang, and Xuan Wang. The dilemma between deduplication and locality: Can both be achieved? In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*, pages 171–185, 2021.

A Artifact Appendix

Abstract

The artifact is source code of a prototype deduplication system for backups that follows the ideas in the paper.

Scope

It could suggest the details and effectiveness of the delta selector, the delta-friendly data layout, the "Always-Forward-Reference" traversing, and the delta prewriting mechanism.

Contents

The artifact is source code of a prototype deduplication system for backups that follows the ideas in the paper. It mainly supports two main operations: (1) deduplicating and storing backup workloads and (2) restoring stored backup workloads.

Detailed manuals are introduced in our GitHub repository. In brief, the artifact supports the two operations with the following two commands.

```
================================================
# deduplicating and storing a new backup
./MeGA --ConfigFile=[config file path] --task=write --InputFile=[backup workload] --DeltaSelectorThreshold=[Delta Selector Threshold]

# restoring a stored backup
./MeGA --ConfigFile=[config file path] --task=restore --RestorePath=[path to restore] --RestoreRecipe=[which backup to restore (1 ∼ n)]
================================================
```

MeGA generates several outputs when executing. Note that:

① MeGA includes chunk-level deduplication, delta compression, and local compression. The "total reduction ratio" suggests the benefits from all these parts on a single backup.

② The "total reduction ratio" simply indicates how many times the size of a single backup has been reduced. For the entire dataset, the user needs to add up the original size of all backups in a dataset and divide it by the "After Compression" of all backups to get the general "Dedup ratio" of the dataset, which is suggested in Figure 15.

③ The backup speed is related to the results in Figure 11.

④ The cache misses and average time cost are related to the results in Figure 12.

⑥ The arrangement duration is related to the results in Figure 17.

⑦ The restore speed is related to the results in Figure 13.

⑤ Figure 16 is just a general view, and it does not have new results.

Hosting

The source cost is available at https://github.com/Borelset/MeGA (the "ContainerBased" branch).

Requirements

The Artifact has the following requirements.
Hardware Requirement:

- CPUs supporting AVX2 instructions.
- 32GB or larger RAM
- 7200rpm HDD drivers for experiments.
- Another storage device for datasets. (400GB for full evaluations or 100GB for partly evaluations)

Software Requirement:

- isal_crypto [https://github.com/intel/isa-l_crypto]
- jemalloc [https://github.com/jemalloc/jemalloc]
- openssl [https://github.com/openssl/openssl]
- zstd [https://github.com/facebook/zstd]
- Reformat your HDD and deploy an XFS file system, as fragmentation of the file system will affect performance.