The Dilemma between Deduplication and Locality: Can Both be Achieved?

Xiangyu Zou and Jingsong Yuan, *Harbin Institute of Technology, Shenzhen;* Philip Shilane, *Dell Technologies;* Wen Xia, *Harbin Institute of Technology, Shenzhen, and Wuhan National Laboratory for Optoelectronics;* Haijun Zhang and Xuan Wang, *Harbin Institute of Technology, Shenzhen*

https://www.usenix.org/conference/fast21/presentation/zou

This paper is included in the Proceedings of the 19th USENIX Conference on File and Storage Technologies.

February 23–25, 2021

978-1-939133-20-5

Open access to the Proceedings of the 19th USENIX Conference on File and Storage Technologies is sponsored by USENIX.

重复数据删除和数据局部性之间的困境：两者都能实现吗？

The Dilemma between Deduplication and Locality: Can Both be Achieved?

Xiangyu Zou[†], *Jingsong Yuan*[†], *Philip Shilane*[*], *Wen Xia*[†‡], *Haijun Zhang*[†], *and Xuan Wang*[†]
† Harbin Institute of Technology, Shenzhen ∗ Dell Technologies
‡ Wuhan National Laboratory for Optoelectronics
Corresponding author: xiawen@hit.edu.cn

Abstract

具体而言，虽然数据去重可以有效减小备份数据的存储需求，但它可能会导致数据在物理存储中不再是相邻的，从而破坏了数据局部性。这种缺乏相邻性可能会影响到数据的恢复过程和垃圾收集的性能，因为需要在不同的内存位置之间进行跳跃，增加了访问成本。这就是所谓的“碎片化问题”或“较差的数据局部性”。

Data deduplication is widely used to reduce the size of backup workloads, but it has the known disadvantage of causing poor data locality, also referred to as the fragmentation problem, which leads to poor restore and garbage collection (GC) performance. Current research has considered writing duplicates to maintain locality (e.g. rewriting) or ~~caching data in memory or SSD~~, but fragmentation continues to hurt restore and GC performance.

数据局部性

Investigating the locality issue, we observed that most duplicate chunks in a backup are directly from its previous backup. We therefore propose a novel management-friendly deduplication framework, called MFDedup, that maintains the locality of backup workloads by using a data classification approach to generate an optimal data layout. Specifically, we use two key techniques: Neighbor-Duplicate-Focus indexing (NDF) and Across-Version-Aware Reorganization scheme (AVAR), to perform duplicate detection against a previous backup and then rearrange chunks with an offline and iterative algorithm into a compact, sequential layout that nearly eliminates random I/O during restoration.

Evaluation results with four backup datasets demonstrates that, compared with state-of-the-art techniques, MFDedup achieves deduplication ratios that are 1.12× to 2.19× higher and restore throughputs that are 2.63× to 11.64× faster due to the optimal data layout we achieve. While the rearranging stage introduces overheads, it is more than offset by a nearly-zero overhead GC process. Moreover, the NDF index only requires indexes for two backup versions, while the traditional index grows with the number of versions retained.

大概意思就是为了兼得数据去重和数据局部性，我们提出了一个新计算模型MFDedup

1 Introduction

Deduplication is an important data reduction technique in modern commercial backup systems because it usually achieves a high deduplication ratio (the logical size divided by the post deduplication size), which was found to often be in the range of 10 ∼ 30× [6], thus greatly reducing storage costs. The basic technique of deduplication is to replace redundant chunks of data with references to identical chunks that have already been stored [39]. While deduplication has been applied to numerous storage and networking topics [6, 12, 21, 29, 45], our research focuses on hard-drive based deduplication for backup storage because it remains one of the most significant use cases.

虽然重复数据删除已经应用于许多存储和网络主题[6,12,21,29,45]，但我们的研究集中在基于硬盘的重复数据删除，因为它仍然是最重要的用例之一

碎片化

For deduplication on hard drive systems, fragmentation is a serious problem: chunks from a backup that are logically consecutive may refer to previously written chunks scattered across the disks (poor locality). As a result, this fragmentation problem causes: ① poor restore performance since many random disk reads are required; ② Garbage Collection (GC) of deduplicated systems is also challenging because as previous backup versions are deleted, referenced and unreferenced chunks may be located together, and referenced chunks must be preserved to avoid data loss.

一段连续的chunk的各个部分，有可能分别指向分散的其他磁盘块

Generally, the root cause of this fragmentation (or poor locality) problem is the sharing of chunks between backup versions due to deduplication. Deduplication systems usually group the deduplicated chunks into a large unit called a **container** (often 4MB in size) for compression and maximizing write performance to arrays of disks and use a chunk-reference list (e.g. a recipe) to record referenced chunks for each backup version. As an example, consider backup *version* 1 that has few or no duplicates, so its chunks are stored sequentially in containers. Then, *version* 2 may be highly redundant with the first with small modifications throughout the backup, so its recipe has references to many chunks of the first version intermixed with references to newly written chunks. Later, *version N* tends to have even worse locality as it refers to chunks written by many previous backup versions, so restoring a backup version involves random seeks back and forth across the disks, and read amplification is high since an accessed container may have needed and unneeded chunks.

这段很生动地解释了reduplication遇到的问题

指在存储系统中读取数据时所引起的数据扩展或数据放大的现象。这通常涉及读取数据时涉及到的额外开销，可能导致实际需要读取的数据量与最终读取的数据量之间存在差异。

To alleviate the fragmentation problem for better restore performance, many techniques have been proposed that write some duplicates (called rewriting approaches) according to their 'fragmentation degree' to maintain a level of data locality [14, 22, 23, 30, 31]. Alternatively, there have been proposals to use memory or SSDs to cache the fragmented chunks or frequently referenced chunks [2,25], which also helps achieve a higher restore speed, though with increased hardware costs. However, fragmentation inevitably becomes worse with high generation backup versions. According to our experimental observations on four backup datasets (see Figures 7 and 9 in Section 5), even using the state-of-the-art rewriting techniques of Capping [23] and HAR [15], the restore speed drops to about 1/8$\sim$1/3 of the sequential read speed of storage devices while the actual deduplication ratio drops about 20% $\sim$ 40% due to 'rewriting'.

The performance of GC is also impacted by data locality in traditional **container-based** data layouts. As older backup versions are deleted, some chunks become unreferenced by any version and can be removed from containers to reclaim space. Generally, GC includes two stages: selecting which containers have unreferenced chunks and migrating referenced chunks into new containers so selected containers can be freed. Several approaches [17,37] have explored ways to quickly select containers for the first stage. When locality is poor, containers will have a mix of referenced and unreferenced chunks, so the migration stage is time-consuming because many chunks must be read and rewritten.

Overall, existing solutions for improving restore and GC performance struggle with the dilemma between deduplication and the locality of backup workloads. Meanwhile, previous work on fragmentation in non-deduplicated storage usually reorganized data to improve the layout [18]. However, due to chunks being shared between backup versions (a complex chunk reference relationship), it seems infeasible to design an optimal layout for all backup versions while maintaining the space savings of deduplication. Moreover, reorganizing chunks can be expensive since some chunks may be referenced by all or most versions: in this case, reorganizing one chunk means almost all versions are involved [23].

In our observations of deduplicating backups, we find that almost all the duplicate chunks in a backup version B_{i+1} are derived from its previous version B_i (studied in Section 3.3), which suggests it is feasible to design *an optimal data layout of deduplicated chunks with nearly no fragmentation*, as explained with three points: ① This **optimal data layout** classifies chunks into categories (like containers) according to their reference relationship. For example, the chunks (a set M), referenced and only referenced by backup versions B_i and B_j, are classified into one category, where a category is similar to a variable-sized container. Classification ensures that if a chunk is required when restoring a backup version, other chunks in the same category are also required, which means loading an entire category does not cause read amplification. ② However, the number of categories (containers) can grow dramatically to about 2^n categories for n backup versions, according to our observation and theoretical analysis. ③ With the above observation that some rare chunk-reference relationships can be ignored, so we only consider chunks that are referenced by one version or by consecutive versions. In this way, the chunk-reference relationship is simplified and the number of categories is reduced to $n(n+1)/2$ for n backup versions, which makes the OPT data layout feasible.

Note that this classification-based OPT data layout is significantly different from a traditional deduplication framework. Traditional deduplication mainly focuses on the write path of deduplication and rarely manages the location and placement of chunks, which we call **write-friendly**. In contrast, our approach tries to redesign the data layout of deduplicated chunks to eliminate the fragmentation problem and thus achieve dramatically faster restore and GC performance, which we call **management-friendly**. In our implementation of the OPT data layout using an offline method of iteratively arranging chunks for each incoming backup version, we find the costs are acceptable, especially compared with the huge overheads of restore and GC in previous write-friendly approaches.

意思就是传统方法容器内的数据块可以按照顺序直接写入，以简化写入过程。而我们的新方法不是直接顺序写入，是按类别写入

To this end, we propose a novel Management-Friendly Deduplication framework, called MFDedup, that introduces two new techniques: Neighbor-Duplicate-Focus indexing (NDF) and Across-Version-Aware Reorganization scheme (AVAR). Together, they generate and maintain the OPT data layout, which eliminates the fragmentation problem. Specifically, the contributions of this paper are three folds:

- We propose NDF to only detect duplicates of a backup version (B_{i+1}) with its previous version (B_i), which utilizes our observation, and provides an opportunity to build the OPT data layout. NDF significantly reduces the memory footprint for the fingerprint index while achieving a near-exact deduplication ratio.
- After deduplicating the new version B_{i+1} using NDF, AVAR arranges the unique chunks of B_{i+1} into the OPT data layout by classifying and grouping according to the simplified reference relationship between chunks and versions. By iteratively updateing the OPT layout, GC becomes a simple operation of immediately deleting the oldest categories as the oldest versions are deleted.
- Evaluation results with four backup datasets suggest that, compared with previous approaches, MFDedup achieves a significantly higher deduplication ratio (1.66× to 3.28× higher) and restore throughput (2.63× to 11.64× higher). Restore throughput fully utilizes the storage devices. Meanwhile, the NDF index is a fixed and limited overhead compared with traditional global index, and GC in MFDedup has nearly zero-overhead.

2 Background and Related Work

2.1 Background of Data Deduplication

Data deduplication is a widely used data reduction approach for storage systems [8,12,27,33,34,38,44,45]. In general, a typical data deduplication system splits the input data stream (e.g., backup files, database snapshots, virtual machine images, etc.) into multiple data "chunks" (e.g., 8KB size) that are each uniquely identified with a cryptographically secure hash signature (e.g., SHA-1), also called a fingerprint [29,34].

在传统的数据去重算法中，相似的数据块或数据片段通常被替换为一个共享的引用或指纹，以减少存储空间的使用。然而，这种操作可能会导致数据局部性的降低，因为原本在物理存储上相邻的数据块可能被替换为分散存储的引用。
当数据在物理存储上分散时，可能会导致数据访问的不连续性，从而降低程序运行时对数据的访问效率。这会对性能产生影响，尤其是对于需要频繁访问相似数据的应用程序。因此，在设计数据去重算法时，需要权衡存储节省和数据局部性之间的关系，以满足特定应用场景的需求。

Deduplication systems then deduplicate data chunks according to their fingerprints and store only one physical copy to achieve the goal of saving storage space.

Backup storage and locality. *Backup storage* often leverages data deduplication due to the highly redundant nature of the data. In backup storage systems, workloads usually are a series of backups versions (i.e., successive snapshots of the primary data), and the size of backups can be greatly reduced to about 1/10-1/30 of their original size, reducing hardware costs. *Locality* in backup workloads means that the chunks of a backup stream will appear in approximately the same order in each full backup with a high probability, which is widely exploited for improving deduplication performance, such as for fingerprint indexing, restoring, etc., by utilizing the high sequential I/O speed of HDDs.

Container-based I/O. Many deduplication-based storage systems usually combine with compression techniques, and all chunks are stored in containers as the basic unit for compression. Thus, storage I/O are usually based on containers. Usually, containers are immutable and have a fixed size (e.g., 4MB). Containers offers several benefits: ① Writing in large units achieves the maximum sequential throughput of hard drives and is compatible with striping across multiple drives in a RAID configuration. Hard drives remain significantly cheaper than SSDs and other media, and cost is an important consideration for backup storage. ② The locality of data in containers is frequently leveraged to improve the efficiency of identifying duplicates as well as for restoring backups to clients [45].

Fragmentation Problem. Fragmentation in deduplication systems is related to container-based I/O and the seek latency of HDDs. This is because different backups will share chunks, and these shared chunks are randomly distributed across containers. In other words, *spatial locality* of each backup will be destroyed after deduplication. Due to container-based I/O, when we restore a backup, even if only a few shared chunks in a container are required, we have to read the whole container from HDDs, which is sometimes called *read amplification* (defined as $\frac{Total\ Size\ of\ Loaded\ Containers}{Size\ of\ Actually\ Restored\ Data}$ during restores). Even if a system supports compression regions within a container, a full compression region must be read and decompressed to supply a needed chunk. In addition, since the required containers for each backup are randomly distributed across the HDDs, *seeking* to these required containers on HDDs is also time-consuming. Moreover, the read amplification and seek issues become worse as the number of backups increases.

2.2 Deduplication Techniques

A typical deduplication system usually consists of several techniques, including chunking, fingerprint indexing, restore optimizations, garbage collection, etc.

Chunking Techniques. Content-Defined Chunking (CDC) [13, 29, 32, 41, 42] is a widely used chunking approach to split the backup stream into variable-sized chunks according to the content, which can handle the 'boundary-shift' problem existing in Fix-Sized Chunking [34].

Fingerprint Index Techniques. Checking the fingerprint index (i.e., detecting duplicates) is a critical step in the workflow of deduplication. Fingerprint indices grow as a fraction of backup storage system capacity, so keeping them in memory is expensive and impractical while putting them in HDDs will cause a deduplication system bottleneck for indexing. Several approaches [5, 7, 17, 24, 26, 28, 40, 45] have been proposed, and most leverage spatial or temporal locality by using the fingerprint index to load many fingerprints from disk that were written at the same time or consecutively in a file.

Restore Optimization Techniques. As introduced in Section 2.1, fragmentation introduces read amplification and many disk seeks when restoring a backup after deduplication. Among the restore optimization approaches, there are two main approaches to review that can be used separately or together: '*rewriting*' and '*cache*'. Rewriting trades-off deduplication space savings to improve locality by selectively writing duplicates [9, 10, 14, 22, 23, 30, 31]. Rewriting lowers the deduplication ratio, and results show read amplification remains $2\times \sim 4\times$ after rewriting. The caching approach uses SSDs or memory to cache chunks that are frequently referenced or believed to be needed in the near future [2, 25], but the cache hit ratio still depends on locality, and read amplification is not addressed.

Among rewriting approaches, Capping [23] follows a simple policy. When deduplicating against a previously-written container, record how many chunks in the container are referenced for the current backup. For containers with low reuse, it rewrites chunks to improve the locality of the current backup version. HAR [15], utilizing the similarity of backup streams, identifies sparse containers according to historical information, and rewrites chunks which refer to those containers. In contrast, our approach writes minimal duplicate chunks and creates a data layout without any fragmentation or read amplification.

Garbage Collection Techniques. Customers usually configure a retention policy for backup files using their backup software, which often involves retaining weeks or months of backups and deleting backups older than the retention policy. GC then removes unreferenced chunks from the system [11, 14, 17, 37]. There are generally two kinds of GC in deduplicated systems. The first one is traditional Mark-Sweep [11, 17, 37]: it walks the backups and marks the chunks referenced from those backups, and then the unreferenced chunks are swept away. In practice, this often requires copying live chunks from a partially-unreferenced container and forming new containers. Although numerous optimizations have been proposed [11, 14, 17, 37], copying live chunks and writing new containers is I/O intensive [15].

The second approach is the Container-Marker Algorithm (CMA) [15]. CMA maintains a container manifest to record referenced backups for each container and then deletes the whole containers that are unreferenced, which is a coarse-grained GC approach that maintains containers if any chunks are still referenced and thus causes wasted space. In contrast, we focus on fine-grained GC, but have an approach with dramatically lower overheads than previous techniques.

3 Observation and Motivation

3.1 Analysis for Fragmentation and Read Amplification after Deduplication

As discussed in Section 2.1, fragmentation causes serious read amplification in deduplication systems using container-based I/O. In this subsection, we analyze the cause of read amplification with a detailed example.

4

3

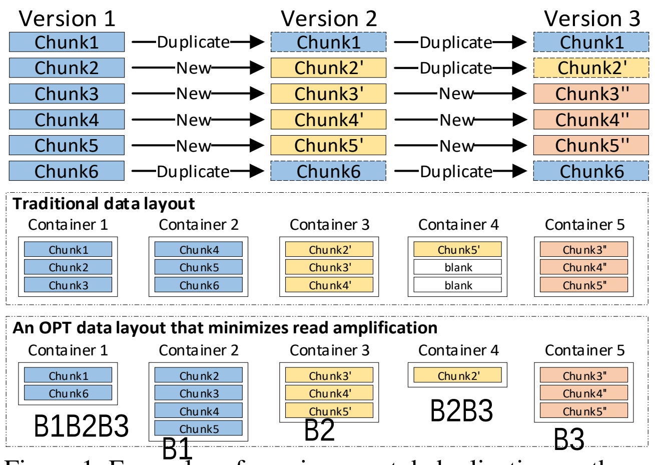


Figure 1: Examples of running exact deduplication on three backup versions with a traditional data layout versus an optimal data layout based on classification.

In traditional deduplication systems, after deduplication, all remaining chunks are stored in containers in the order they appear in a backup. Figure 1 shows an example of the traditional data layout after deduplication of three backup versions. Deduplicated chunks from three backup versions are stored in five containers using the traditional data layout, and *Chunk* 6 is referenced by all the three versions. Because of the container-based I/O, no matter which version we want to restore, we always need to read *Container* 2 from HDDs.

Read Amplification. For *Versions* 2 and 3, only *Chunk* 6 is needed from *Container* 2, which includes other chunks unreferenced by *Versions* 2 and 3. This is an example of poor spatial locality. Therefore, loading *Container* 2 causes read amplification (i.e., we read two unneeded chunks) when restoring these two versions. But for *Version* 1, all chunks in *Container* 2 are required, which means a strong spatial locality, and there is no fragmentation and read amplification. Note that under the traditional data layout, *Chunk* 6 is a fragmented chunk for *Versions* 2 and 3, but is not a fragmented chunk for *Version* 1, which means fragmentation is dependent on the backup version and associated with chunks' reference relationship.

3.2 An Optimal Data Layout

In this subsection, we present and discuss a classification-based optimal (OPT) data layout according to the chunks' reference relationships as mentioned in the last subsection.

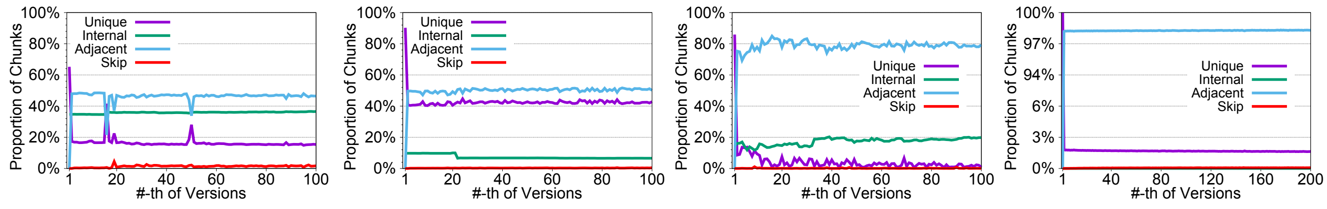
An Example of Classification: For the three chunks {4, 5, 6} in *Container* 2 (in the traditional data layout in Figure 1), according to their reference relationship, we can classify them into two categories (like containers). The first category includes *Chunks* 4 and 5, which are only referenced by *Version* 1, and the second category is *Chunk* 6, which is referenced by all three versions. If we store two categories separately in different containers, we could load both categories when restoring *Version* 1 and load only the second category for *Versions* 2 and 3. In this way, the fragmentation problem of these three chunks are resolved, and there will be no read amplification when restoring any of the versions.

Classification-based Data Layout. Here we continue the previous example and classify all chunks into five categories (like containers) according to their reference relationship and then *store each category into a variable-sized container*, as shown in the 'OPT data layout' in Figure 1:

- *Container* 1 is referenced by Versions {1, 2, 3}.
- *Container* 2 is referenced by Version 1.
- *Container* 3 is referenced by Version 2.
- *Container* 4 is referenced by Versions {2, 3}.
- *Container* 5 is referenced by Version 3.

This layout keeps strong spatial locality for each backup version with a read amplification of 1, so we refer to it as the OPT data layout that minimizes read amplification. For example, if we want to restore *Version* 3, we need to load *Containers* $1, 4$ *and* 5, which does not load any unrequired chunks. Meanwhile, there is also no read amplification when restoring *Versions* 1 and 2. Therefore, Figure 1 provides a possible solution to eliminate fragmentation for that example of three backup versions. In the worst case of three backup versions, there will be seven categories (i.e., total number of $\binom{3}{1} + \binom{3}{2} + \binom{3}{3} = 2^3 - 1$ reference relationship). Here $\binom{n}{k}$ means choosing k from n elements.

Challenges for OPT Data Layout. Actual backup workloads are much more complicated than the example shown in Figure 1. Specifically, if we follow the idea of classification on n backup versions, there will be $2^n - 1$ (i.e., $\binom{n}{1} + \binom{n}{2} + ... + \binom{n}{n} = 2^n - 1$) categories that are stored as $2^n - 1$ containers. Assuming there are 30 backup versions with about 1 million unique 8KB chunks (totaling 8 GB after deduplication), there will be more than 1,000,000,000 containers after classification and thus most of the time each container has only one or very few chunks. In other words, this OPT data layout solves the read amplification problem but requires more seek operations for these very small containers, which also causes poor data locality.



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

Figure 2: Distribution of four kinds of chunks on four backup datasets. *Skip* duplicate chunks are the least common.

3.3 Derivation Relationship of Backups

In this subsection, we will present our key observation about the deduplication relationship of backups through an analysis of four large backup datasets. These relationships can be exploited to greatly decrease the number of categories (i.e., containers) needed for the OPT data layout.

In backup storage systems, workloads usually consist of a series of backup images, which are all generated from the original data on a primary storage system (i.e. laptop, server, database, etc.). Therefore, the duplicate chunks of each backup are not randomly distributed but are derived from the chunks of the last backup as we will show with experiments, which is consistent with the typical consecutive pattern of duplicates that is leveraged by many systems [24, 40, 45] for high deduplication performance.

To better illustrate our observation, we denote four kinds of chunks in a backup version B_i as follows:

- *Internal duplicate chunks*, whose referenced chunks are also in B_i.
- *Adjacent duplicate chunks*, whose referenced chunks are not in B_i but in the last version B_{i-1}.
- *Skip duplicate chunks*, whose referenced chunks are neither in B_i nor in B_{i-1}.
- *Unique chunks*: the non-duplicate chunks.

Key Observation. Figure 2 studies the distribution of the four kinds of chunks on four backup datasets running with exact deduplication. From Figure 2, we can observe that most duplicate chunks for a backup version are from the previous version (*Adjacent*) and within the current backup itself (*Internal*). Adjacent and Internal account for more than 99.5% in most datasets. The *Skip* duplicate chunks only consist of a small fraction (less than 0.5% in most datasets) of all duplicate chunks. This observation supports an approach of avoiding deduplicating Skip chunks to preserve locality since they would only have a small impact on the deduplication ratio.

Motivated by the above observation of the duplicate chunks' pattern, we avoid deduplicating Skip chunks and treat them as Unique chunks for the current version. This greatly simplifies chunk-reference relationships: each physical chunk must be referenced by one version or by consecutive versions (e.g., B_i, ... B_{i+k}) in the former OPT data layout. With this condition, we can greatly reduce the number of classified categories (containers). Taking three backups for example, because each chunk must be referenced by successive versions $\{B_i, ..., B_{i+k}\}$, where $i \geq 1$ and $i+k \leq 3$. Thus, when k=0, there are $\binom{3}{1}$ categories; when $k \neq 0$, there are $\binom{3}{2}$ categories (choosing the start and the end one for successive versions). Hence, the number of categories in this example is reduced from seven to six.

In general, if we have n backup versions and improve the OPT data layout by exploiting the duplicate chunks' pattern, the upper limit of the number of categories (containers) will be $\binom{n}{1} + \binom{n}{2} = n(n+1)/2$, which is much less than $2^n - 1$. Continuing an earlier example with 30 backup versions, there will be 465 containers with about 2150 chunks on average, so the average size of container is about 17.6MB. This condition makes the classification feasible for the OPT data layout while the size of containers is large enough to maintain the spatial locality of backup workloads.

4 Design and Implementation

4.1 MFDedup Overview

Based on our key observations about the relationships between backups and the OPT data layout mentioned in Section 3, implementing the optimal data layout in a deduplicated backup storage systems is feasible by the following two key design principles: ① *All chunks are classified into categories (like containers) according to their reference relationship.* ② *Skip duplicate chunks are treated as unique chunks to simplify the chunks' reference relationship.*

In this paper, we propose our approach MFDedup, a management-friendly deduplication framework using the aforementioned optimal (OPT) data layout. Our approach is to maintain the locality of backup workloads, which eliminates fragmentation that slows restore and GC. MFDedup follows the above two design principles and reorganizes chunks in an offline algorithm to achieve the OPT data layout, by using two key techniques:

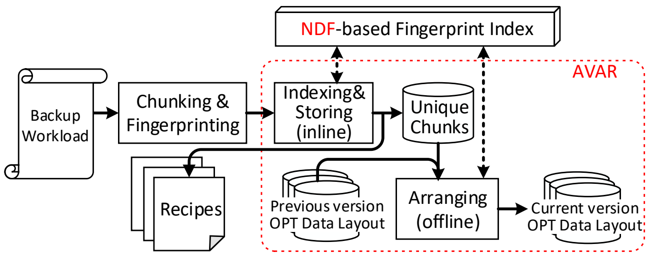
- **Neighbor-Duplicate-Focus indexing (NDF)**. MFDedup only removes duplicates between neighboring backup versions. Hence, we only need to build and access a local fingerprint index consisting of the neighboring backup versions' fingerprints, whose resource requirements are lower compared with traditional global fingerprint index, as detailed in Section 4.2.

确实感觉只用跟
最邻近的版本对
比就行。
如果重复，那么
则加入所有含有
Vi-1Vi的
category；否则
单独加一个Vi的
category

- **Across-Version-Aware Reorganization (AVAR).** After detecting duplicate chunks of each new backup ver-

感觉跟nelcm的思路也是差不多的，都是guide和实际的arrange解耦了
只不过mfdedup的arrange新弄了个阶段，nelcm在gc一起做了
不过想想为什么mfdedup的arrange需要一个新的stage，nelcm不用？

sion using NDF, MFDedup offline arranges the deduplicated chunks of the last backup version. Specifically, these chunks are classified and grouped to iteratively update the OPT data layout according to our simplified chunk-reference relationship, as detailed in Section 4.3.



NDF-based Fingerprint Index
AVAR
Backup Workload
Chunking & Fingerprinting
Indexing& Storing (inline)
Unique Chunks
Recipes
Previous version OPT Data Layout
Arranging (offline)
Current version OPT Data Layout

Figure 3: An overview of MFDedup framework.

The overall workflow of the MFDedup framework is shown in Figure 3, which includes three key stages: *Chunking & Fingerprinting, Indexing & Storing, and Arranging*. Chunking & Fingerprinting refers to splitting the backup stream into chunks using Content-Defined Chunking [29, 39] and then calculating a fingerprint (i.e. SHA1 digest) for each chunk. Indexing & Storing detects duplicate and unique chunks from the previous backup version by using NDF-based fingerprint index and then stores unique chunks and a **Recipe** for each backup. Arranging is an offline process, which iteratively updates the OPT data layout version by version, with the support of NDF-based fingerprint index. Note that the Recipe records the chunk-fingerprint sequence of a backup version, which is used to recover the backup version after deduplication.

In general, MFDedup applies online deduplication using NDF, which removes duplicates only between neighboring backup versions, and then an offline Arranging using AVAR, which keeps the OPT data layout to maintain locality of backup workloads and thus eliminate fragmentation.

4.2 Neighbor-Duplicate-Focus Indexing

In this section, we will introduce the Neighbor-Duplicate-Focus indexing (NDF) technique in MFDedup, which is based on our observation (Section 3.3) that most duplicate chunks exist between neighboring versions in a backup storage system (i.e., duplicate chunks of backup version B_i are nearly all from its previous version B_{i-1}). Therefore, MFDedup chooses to treat *Skip* duplicate chunks as unique chunks instead of deduplicating them. In other words, MFDedup only identifies duplicate chunks in backup version B_i that are identical to chunks either in B_i (within the same version) or B_{i-1} (the previous version). We refer to this as a **NDF-based fingerprint index**.

In the NDF implementation, we maintain an independent fingerprint index table for each backup version. Besides using NDF in the Indexing & Storing stage of MFDedup for duplicate detection, NDF is also used in the *Arranging* stage for classification as detailed in next subsection. After a finger-

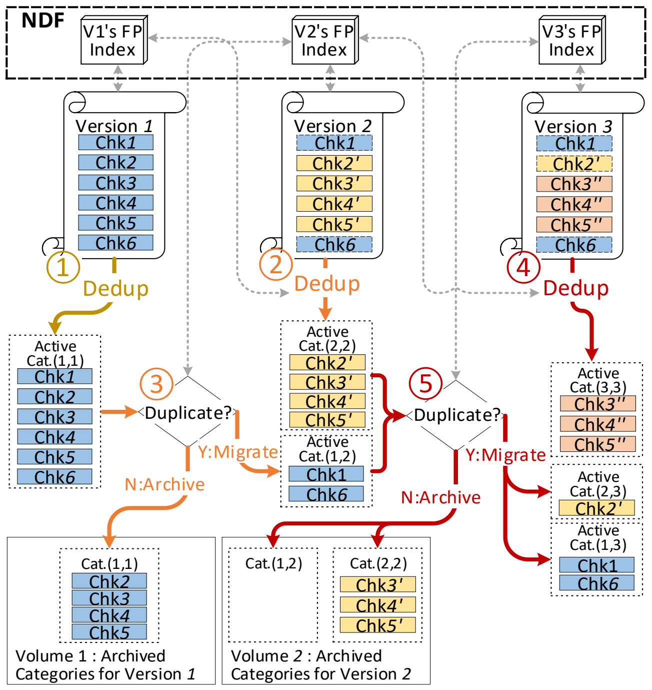


Figure 4: An example of the AVAR workflow on three backup versions, which is presented by a solid line in five steps: ① *Deduplicating* Version 1 → ② *Deduplicating* Version 2 → ③ *Arranging* Version 1 → ④ *Deduplicating* Version 3 → ⑤ *Arranging* Version 2. Gray dashed lines refer to fingerprint indexing operations.

print index table is used in the above two stages for the latest two backup versions, it can be released. Therefore, we only need to maintain two fingerprint indices, which could be kept in memory if they are small (they are typically much smaller than the traditional index that stores all versions) or loaded using previous locality-based approaches [24, 45]. Assuming a fingerprint index entry takes 20 bytes (i.e., the size of SHA1 digest), the size of a backup version is 10GB, and the expected chunk size is 8KB, the total memory cost of NDF-based fingerprint index will be $2 \times 10GB/8KB \times 20B = 50MB$ (about 0.4882% size of a backup version).

Indexing overhead of NDF is related to the data size of two most recent backup versions, which is considerably smaller than traditional deduplication systems that have a global fingerprint index. Meanwhile, NDF is able to achieve a near-exact deduplication ratio while supporting the OPT data layout in MFDedup (detailed in Section 4.3).

4.3 Across-Version-Aware Reorganization

In this subsection, we will introduce Across-Version-Aware Reorganization (AVAR) in MFDedup, which is designed to eliminate fragmentation and generate the OPT data layout combining with the NDF technique.

There are two stages in AVAR, which are the *Deduplicating* stage (i.e., *Indexing & Storing* in Section 4.1) and the *Arranging* stage, and both utilize the NDF-based fingerprint index.

Figure 4 gives an example of AVAR on three backup versions, running with the two stages alternating (except the first backup version): The Deduplicating stage identifies unique chunks of a backup version B_i. Then the *Arranging* stage updates the OPT data layout for backup version sets $\{B_1..B_i\}$, by reorganizing the previous OPT data layout of $\{B_1..B_{i-1}\}$ with the unique chunks of B_i. Note that there is naturally an OPT data layout when the first backup version is stored, so the Arranging stage is not required after the Deduplicating stage. More details about the two stages of AVAR are elaborated below.

Deduplicating* Stage.** In this stage, we detect duplicate chunks using the NDF-based fingerprint index and then store unique chunks as well as Recipes. In the remainder of this section, we ignore Recipes, and focus on how data chunks are managed for the OPT data layout. Therefore, as shown in Steps ①, ② and ④ of Figure 4, the Deduplicating Stage is responsible for storing unique chunks of the latest new backup version B_i into a new active ***Category, which is currently only referenced by B_i. Note that chunks in the active Category may be referenced by future backup versions and thus will be processed by the Arranging stage later. 确实，arrange需要处理Bi-1的unique chunk被Bi引用的情况

Note that each Category is named with a pair of numbers in MFDedup, which reflects which backup versions refer to chunks in this category. For example, if chunks in a category are referenced from consecutive Versions 2, 3, and 4, we denote this category as **Cat.(2, 4)**.

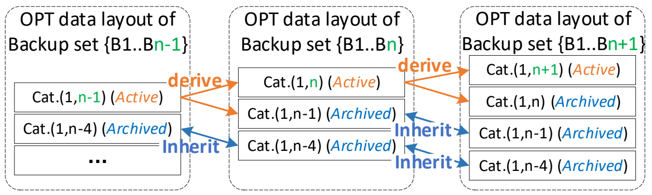
***Arranging* Stage.** According to the 1st principle of MFDedup, classification methods used for generating the OPT data layout are based on the reference relationship between chunks and backup versions, which means the old OPT data layout expires when a new version arrives and is processed by the *Deduplicating* stage. This is because the reference relationship between chunks and backups has changed. Therefore, the Arranging stage is responsible for iteratively updating the existing OPT data layout with new unique chunks of the incoming version (after the Deduplicating stage), following the two design principle of MFDedup.

看这意思，arrange是每次ingest后马上做的？

To better present the iterative process of our Arranging stage, Figure 5 shows a general evolution example of OPT data layout with three backup versions. According to the design principle ② in Section 4.1 (i.e., *Skip* duplicate chunks are ignored and treated as unique chunks), in backup version sets $\{B_1..B_{n-1}\}$. Then Cat.(1,n-4) is not referenced by the last backup version B_{n-1} and cannot be referenced by B_n and later backup versions. As a result, these kinds of categories are always carried forward as part of the OPT data layout and are referred to as **archived**. On the other hand, Cat.(1,n-1) that is referenced by the last backup version B_{n-1}, will be split into two categories when backing up a new version B_n. We call those categories **active**.

Therefore, in our implementation of AVAR, classified categories (containers) have two states: **Active and Archived** when updating the OPT data layout. Archived means the cat-

确实，这个分裂关系很正确



这个图大概是演示了Arrange的过程，它通过的NDF index省略未画出

Figure 5: An example of the OPT data layout's evolution on three backup versions. Some categories are inherited from the previous version, and some derive new categories.

egories are immutable, while Active means the categories will be further 'arranged' by MFDedup after future backups. More specifically, in the Arranging stage of AVAR, we focus on active categories, which derive new active categories and archived categories. Like the example of *Step ⑤* shown in Figure 4, Cat.(1,2) and Cat.(2,2) are the only two existing (old) active categories after backing up Version 3, and we check each chunk of them with the fingerprint index of backup version 3. Duplicate chunks, existing in Version 3, are migrated to new active Cat.(1,3) and Cat.(2,3), and other chunks are arranged in archived Cat.(1,2) and Cat.(2,2). After migrating and archiving, old active Cat.(1,2) and Cat.(2,2) are no longer required and thus deleted.

Grouping. After Arranging existing Active categories, the new Archived categories are grouped into a **Volume** by the order of their name (e.g. in the order of Cat.(1,3), Cat.(2,3), Cat.(3,3)), for easier storage management. The benefits of grouping categories will be introduced in the next section.

这应该是为了增强restore吞吐

4.4 Restore and Garbage Collection

Restore and Garbage Collection both benefit from our OPT data layout in MFDedup, and their workflows are greatly simplified as elaborated in this subsection.

Restore. When restoring a backup version in MFDedup, we only need to read the required categories on the OPT data layout, which is referenced by the to-be-restored version. Meanwhile, tracing required chunks (categories) for restore is totally metadata-free in MFDedup with the support of the OPT data layout (i.e., can be calculated by our layout).

For the situation that there are n backup versions stored in MFDedup, and we want to restore a backup version B_k, all categories referenced by B_k are required. For example, Cat.(3,k+2) is required, because it is referenced from B_3 to B_{k+2}, which includes B_k. Thus, all required categories for B_k could be represented as:

$$\begin{aligned} Required\ Cat. &= \{Cat.(i,j)\}, where\ 1 \le i \le k \le j \le n \\ &= \cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j). \end{aligned} \tag{1}$$

Like the example of Figure 6, there are four stored backups. According to Equation 1, restoring Version 3 requires the blue-colored categories. Note that according to our grouping approach, $\cup_{i=1}^{j} Cat.(i,j)$ are always sequentially grouped in the same Volume. Thus, loading $\cup_{j=k}^{n} \cup_{i=1}^{j} Cat.(i,j)$ requires n sequential reads at most.

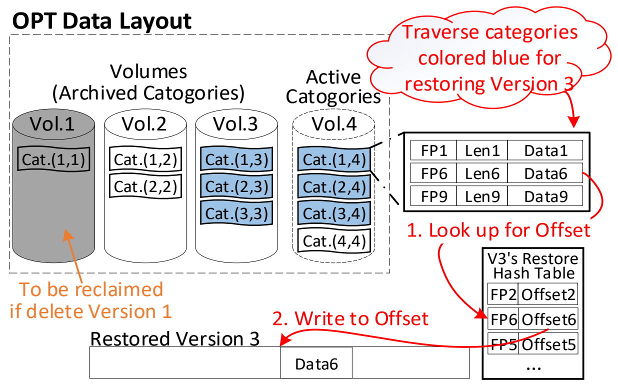


Figure 6: An example of restore and deletion on the OPT data layout with four backup versions.

A recipe is required to restore a backup version, which is used to build a 'restore' hash table, whose format is shown in Figure 6. The entry of the hash table is a pair like <fingerprint, offset>, which records a chunks' fingerprint and its offset in the to-be-restored version.

The workflow of restore is shown in Figure 6, after getting the required categories, the chunks are restored one by one according to the Recipe for Version 3. Therefore, MFDedup only needs to seek to the required volumes and then sequentially read the required (consecutive) categories in those volumes, which achieves a superior restore performance (i.e., few seeks and large sequential I/Os).

Deletion and Garbage Collection. As a result of our OPT data layout, deletion and garbage collection are naturally simple, and the space can be immediately reclaimed in MFDedup. In deduplication systems, deleting a backup version means reclaiming its unique chunks (those not referenced by other backups). FIFO-based deletion in MFDedup simply deletes and reclaims the earliest volumes, because they consist of unique chunks of the earliest backup versions. For example in Figure 6, we can reclaim space of *Version* 1 by directly deleting *Volume* 1.

如果删除最早版本，直接删除第一个volume即可

这很显然，因为第一个volume只有一个cat(1,1)理应被删

MFDedup also supports deleting other backup versions besides the earliest ones. From the description of the Deduplicating stage, we see that the unique chunks of each backup version are always stored in the last category of each volume (see Figure 6). Thus, we can also delete any backup version by resizing the corresponding volume using '*truncate()*' (i.e., deleting the last category in this file). For example, if we want to delete *Version* 2 in Figure 6, we can remove the archived Category 3 by just truncating *Volume* 2. A previous work [11] mentioned that the CMA approach [15] only supports FIFO deletion, while we support any deletion pattern.

如果是随机删除版本，按他的意思感觉是说只需resize一下对应volume即可，其它的为啥不需要调整呢？比如说有个cat(1,4)但是v3被删了，难道其实影响不大吗

好吧确实应该影响不大。只要注意释放那些cat(m,n)且m到n中间都被删了的volume即可

In this way, we no longer apply traditional GC techniques in MFDedup, such as mark-sweep or reference-count management, since the chunk-reference relationship is naturally designed into our classification-based OPT data layout. This is a dramatic reduction in system resources (CPU cycles, RAM, I/O) and coding complexity.

4.5 Discussion and Limitations

In this subsection, we discuss overheads, limitations, and corresponding possible optimizations of MFDedup in a deployed system to support various backup workloads.

Self-Organization of OPT data layout. This OPT data layout is self-organized and simple, and the cost of metadata is greatly reduced. The exact physical position and the reference counts of each unique chunk, which are usually used for restore and GC in traditional deduplication systems, are not needed for MFDedup. For example, in restore, the required categories for each version are calculable in the OPT data layout.

Backups Size. While backup sizes can vary over a wide range, many VM backups are ~100GB, and the index is 400MB for the most recent virtual machines. Wallace [38] and Amvrosiadis [4] also suggested the majority of backups were 50-500GB in Data Domain and Symantec production systems. Hence MFDedup can be directly applied in these scenarios with a reasonable memory overhead.

Fingerprint Prefetching for Larger Backups. Although the current design of MFDedup has the index in RAM, previous techniques for prefetching and caching sequences of fingerprints (designed for a large fingerprint index) [5,7,17,24,26,28,40,45] could also be used in MFDedup. We expect the sequential locality to also exist in MFDedup for two reasons. On the one hand, sequential locality exists inside categories, although it is destroyed across categories by Arranging. On the other hand, recipes also keep the sequential locality of each backup.

Restoring for Larger Backups. When restoring a single large backup, since chunks are organized with categories in MFDedup, we could also organize a 'restore' hash table (recording pairs <fingerprint, offset>) for each category, and then load the hash tables to memory separately. Besides, a single backup could be divided by MFDedup into several smaller sub-units (e.g., each <100GB) to relieve the memory burden for both backing up and restoring. But, when the size of a single backup is huge (such as over $10TB$), and there is only one category for this backup, the hash table would also be very large (over $10TB/8KB \times (20B+8B) = 35GB$, here we assume 'fingerprint' and 'offset' take 20B and 8B, respectively), which is difficult to maintain in memory, so MFDedup can not handle these use cases yet.

Incremental Backups vs. Full Backups As we introduced in Section 4, MFDedup is designed for full backups. For incremental backups, we could add an API to distinguish between incremental and full backups. Also, as synthetic full backups have already become broadly used, the incremental changes are typically relative to the last "full" backup synthesized, so MFDedup can also be directly applied.

Reserved Space for Arranging. Arranging is an offline process, in which chunks are migrated or archived, and it requires additional reserved space. As shown in Figure 4, Arranging runs on active categories, thus, the reserved space is equal to the maximum size of active categories, which is much smaller than a full backup and is studied in Section 5.6.

What if Arranging Falls Behind. If there are a lot of workloads to back up and not enough time to finish the *Arranging* stage in MFDedup, we can skip it temporarily, and apply it in future idle time. Before Arranging catches up, the OPT data layout is not updated with new incoming backup versions. The more Arranging falls behind, the more seriously OPT data layout is damaged, with an increase in read amplification and decrease in restore throughput. However, this a rare case since users usually create full backups daily or less frequently [3, 23], which provides enough time for our offline Arranging. In addition, a higher deduplication ratio leads to a smaller read amplification and also a smaller reduction in restore throughput when Arranging falls behind.

Time Overhead of Offline Arranging. In MFDedup, we have transferred background work from GC to Arranging while achieving many benefits: high restore speed, immediate space reclamation, etc. Arranging is an offline process that traverses active categories of a backup version, migrates duplicate chunks, and archives the remaining chunks. The time cost of Arranging is close to or better than perfect garbage collection with the benefits of better restore and GC performance of MFDedup as evaluated in Section 5.5. In the paper, we always run Arranging after each backup to keep the data layout healthy (i.e., optimal), but Arranging could act like GC: just running once after several backups. In this case, Arranging falls behind and will cause slight read amplification, as discussed in "What if Arranging Falls Behind". Besides, several Arranging tasks, in which duplicate chunks will be migrated several times, could be merged in this situation, which could reduce the total overhead for Arranging, though this has not been evaluated.

Out-of-Order Restore. Unlike the implementation of the traditional deduplication framework, restore in MFDedup is out-of-order, which means the writing order of restored chunks does not absolutely follow their logical order in workloads. While the chunks in volumes are generally in order for a backup, there are logical gaps that are filled by other volumes, which causes random writes to the restored version. Although the sequential locality still exists inside categories, as discussed in "Fingerprint Prefetching", restore will have better performance if the destination media has good random write performance (e.g., on SSDs). Besides, some previous techniques, like a reassembly buffer [20], could be applied to improve the performance when streaming a restore to HDD devices.

确实，这个reassembly buffer可以减少random seek，这个角度我还真没注意到过

5 Performance Evaluation

5.1 Experimental Setup

Evaluation Platform and Configurations. We perform our experiments on a workstation running Ubuntu 18.04 with an Intel Core i7-8700 @ 3.2GHz CPU, 64GB memory, Intel D3-S4610 SSDs, and 7200rpm HDDs.

Table 1: Four backup datasets used in evaluation.

Name	Total Size Before Dedup	Versions	Workload Descriptions
WEB	269 GB	100	Backup snapshots of website: news.sina.com, captured from June to September in 2016.
CHM	279 GB	100	Source codes of Chromium project from v82.0.4066 to v85.0.4165
VMS	1.55 TB	100	Backups of an Ubuntu 12.04 Virtual Machine
SYN	1.38 TB	200	Synthetic backups by simulating file create/delete/modify operations [36]

In our evaluation, we built a MFDedup prototype system and also built Destor [16] for comparison with several state-of-the-art techniques for restore and GC, including the History-Aware Rewriting algorithm (HAR) [15], Capping [23], and Container-Marker Algorithm (CMA) [14]. MFDedup and Destor use the same configuration in the Chunking & Fingerprinting stage: chunking uses FastCDC [41] with the minimum, average, and maximum chunk sizes set to 2KB, 8KB, and 64KB; Fingerprinting uses a SHA1 digest generated by Intel Intelligent Storage Acceleration Library Crypto Version (i.e., ISA-L_crypto [1]).

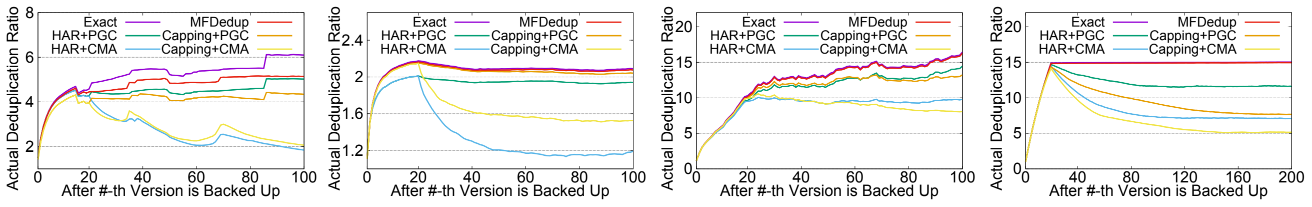
Experimental methods. To simulate real backup/restore scenarios, we separate the storage space of our workstation into two parts: a backup space using a 7200rpm HDD and a user space using an Intel D3-S4610 SSD. Both spaces (drives) have an XFS file system. It is typical in backup environments to use HDDs for cost reasons while primary systems often use SSD for higher performance.

To evaluate backup/restore performance, tested datasets are backed up from the user space to the backup space version by version while the restore runs in the reverse direction. Note that before each backup/restore, we always flush the file system cache using the command: "echo 3 > /proc/sys/vm/drop_caches".

To simulate users' retention (deletion) requirements in backup systems, we retain the most recent 20 versions. Thus Version $n-20$ is deleted after Version n is backed up, which is the same as the previous work HAR [15] and CMA [14]. For throughput (time cost) of backup, restore, and GC/Arranging in our evaluation, we present the average results of five runs.

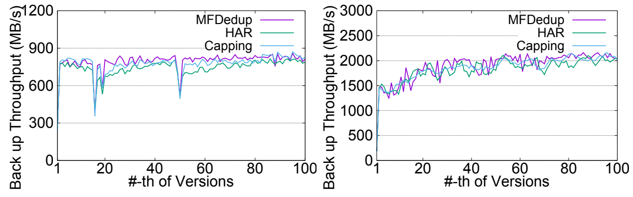
Container-based I/O is considered, because many deduplication-based storage systems usually combine with compression techniques, and all chunks are stored in containers as the basic unit for compression. Because here we focus on deduplication and compression techiniques are orthogonal to deduplication, we do not introduce compression in evaluations.

Evaluation Dataset. Four backup datasets are used for evaluation as shown in Table 1. These datasets represent various typical backup workloads, including website snapshots, an open source code project, virtual machine images, and a synthetic dataset, with deduplication ratios varying from 2.19 to 44.65. WEB, SYN, and VMS datasets have been used in several studies of data deduplication [15, 41, 43].



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

Figure 7: Actual Deduplication Ratio of MFDedup and five approaches running on four datasets (retaining 20 backups).



(a) WEB (b) VMS

Figure 8: Backup throughput of MFDedup and two other rewriting approaches running on two selected datasets (due to space limit), without considering (offline) Arranging.

5.2 Actual Deduplication Ratio

As mentioned in Section 2, rewriting and GC techniques (e.g. HAR, Capping, and CMA) consume more storage space in exchange for better restore and GC performance. Meanwhile, MFDedup ignores Skip duplicate chunks to implement OPT data layout, which also reduces the deduplication ratio. Hence, in this subsection, we evaluate MFDedup and other approaches with the *Actual Deduplication Ratio* (denoted by ADR) defined as $\frac{Total\ Size\ of\ the\ Dataset}{Size\ after\ Running\ an\ Approach}$, which reflects the corresponding reduced deduplication ratio due to these techniques (such as rewriting).

Figure 7 shows ADR of MFDedup, Exact Deduplication, and other approaches, including combinations of rewriting (HAR and Capping) and GC (Perfect GC and CMA) techniques. Here MFDedup includes its GC approach. Note that we only retain the latest 20 backup versions in our evaluation, and thus Perfect GC and CMA represent two typical GC techniques using Mark-Sweep with utilization thresholds set at 0% and 100%, respectively. Perfect GC reclaims all possible space, while CMA runs faster but leaves unreferenced chunks in containers that are partially referenced, so they show two kinds of extreme impacts of GC.

Generally, Figure 7 shows that MFDedup achieves ADR that is very close to Exact deduplication, which is much higher than other rewriting and GC approaches. This is because the space cost of ignoring Skip duplicate chunks in MFDedup is quite small, especially compared with the number of rewritten chunks in other approaches.

Figure 7 also shows rewriting techniques cause a decrease in ADR when GC starts after version 21. When the CMA technique (higher GC speed, fewer unreferenced chunks removed) is added, this loss worsens. This is consistent with our discussion in Section 2: rewriting reduces deduplication while GC also can lead to more rewritten chunks. Meanwhile, deletion and GC are naturally supported in our OPT data layout with NDF and AVAR techniques, which has no fragmentation issue and thus no space cost for MFDedup. Overall, MFDedup achieves a 1.12× to 2.19× higher ADR than other approaches due to the OPT data layout.

5.3 Backup Throughput

In this section, we study backup throughput of MFDedup compared with rewriting approaches. Here we do not consider the impact of GC since it is usually an offline process. Both HAR and Capping use a full-in-memory global fingerprint index while MFDedup applies NDF-based local fingerprint index. To minimize the performance impact of reading datasets, we back up the datasets from a ramdisk to measure the backup throughput.

Figure 8 shows backup throughput of the three approaches, which have similar results for a given dataset. This highlights that MFDedup does not sacrifice backup throughput to achieve the other benefits we discuss. The performance of the three techniques is similarly limited by the chunking and SHA1 digest calculation. In theory, since MFDedup no longer rewrites duplicate chunks (thus achieving higher Actual Deduplication Ratio in Section 5.2), its storage I/O when backing up will also be smaller than the traditional design.

Indexing Overhead. During backups, we measured the maximum memory cost for the NDF index, which varied from 6.27MB to 46.35MB (only indexing 2 backup versions). In contrast, traditional deduplication approaches maintain a global fingerprint index for all 20 backup versions and would require 26.81MB to 64.45MB space. Note that the traditional global index grows with the number of retained versions, while NDF only maintains 2 indices.

5.4 Restore Throughput

Previous approaches [15, 23] use **Speed Factor** to measure restore throughput. It is defined as the ratio of useful data restored per container read in deduplication-based backup systems, assuming using fix-sized containers as the read I/O unit [23]. Since MFDedup uses variable-sized containers to hold categories as the I/O unit, thus we define three metrics in this subsection, **Restore Throughput**, **Seek Number**, and **Read Amplification Factor**. Here Seek Number is defined as the number of seek operations required for reading containers/volumes on disk devices while Read Amplification Factor

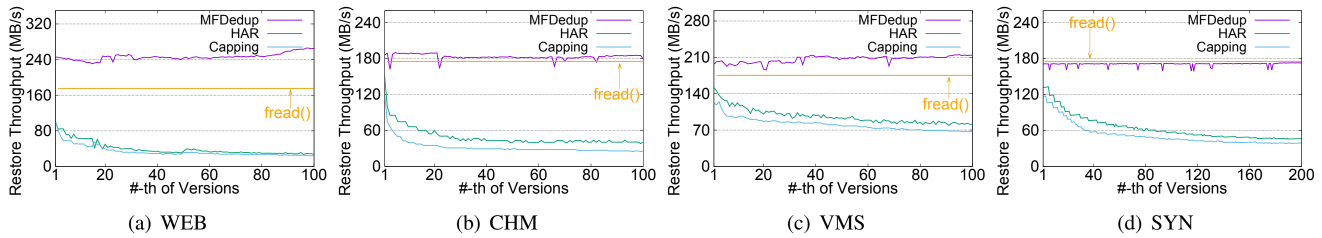


Figure 9: Restore Throughput of MFDedup, HAR, and Capping on four backup datasets. *fread()* denotes sequential throughput of the backup device.

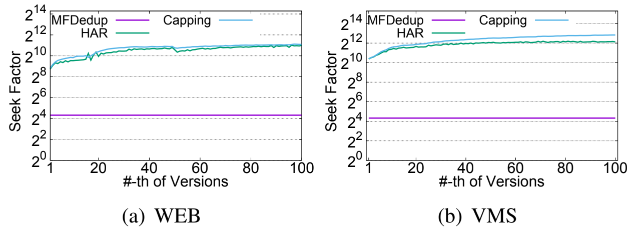


Figure 10: Seek Number of MFDedup, HAR, and Capping on restoring two typical datasets (due to space limit).

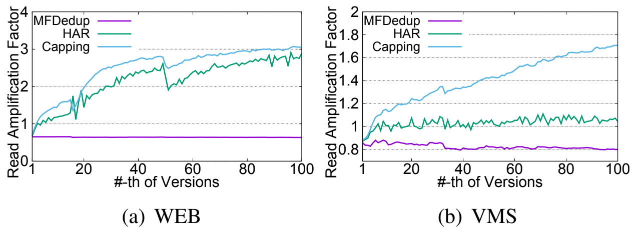


Figure 11: Read Amplification Factor of MFDedup, HAR, and Capping on restoring two datasets (due to space limit).

is defined in Section 2.2.

Figures 9, 10 and 11 present the restore results of MFDedup, HAR, and Capping on the three metrics, which demonstrate that Restore Throughput is generally consistent with the other two metrics. Figure 9 shows that HAR performs better than Capping in Restore Throughput, but MFDedup achieves up to 11.64× (WEB), 4.54× (CHM), 2.63× (VMS) and 3.73× (SYN) higher than HAR. This is because MFDedup has eliminated fragmentation by maintaining locality of backup workloads on the OPT data layout, while fragmentation (though alleviated) still exists in HAR and Capping based systems and becomes worse with higher versions.

Figure 10 shows the Seek Number on two datasets. Results for the other datasets were consistent and removed for space reasons. MFDedup reduces the Seek Number from thousands for HAR and Capping to 20, which is because it groups several archived categories into one big, sequentially written volume; Capping and HAR need more seek operations due to their scattered distribution of required chunks.

On the other hand, Figure 11 shows the Read Amplification Factor (results were consistent for all datasets). MFDedup has the smallest Read Amplification Factor, which is only 34.32% of Capping and 50.19% of HAR on average. This is because its OPT data layout has eliminated fragmentation. Meanwhile, HAR and Capping will encounter more unneeded chunks in loaded containers when restoring. Read Amplification Factor is less than 1 for MFDedup due to Internal deduplication within a backup version (Figure 2), so read chunks can be used multiple times for a restore. Therefore, restore throughput of MFDedup is even higher than the storage media: up to 1.5× of *fread()*, which means MFDedup can completely utilize the performance of storage devices.

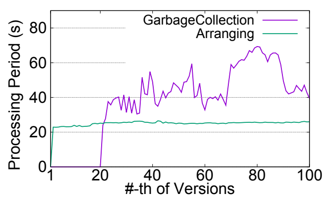
Note that these result are also evaluated while retaining 20 backup versions. If we retain more backup versions, the restore results of HAR and Capping will decrease, as is discussed in many previous works [15, 23]. Without fragmentation, MFDedup achieves a consistently high Restore Throughput, even when retaining more backup versions.

5.5 Arranging vs. Traditional GC

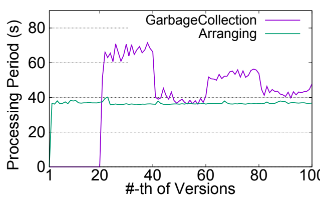
Compared with traditional deduplication approaches, MFDedup has the benefit of nearly zero-overhead Garbage Collection (GC), but adds the offline Arranging process. Therefore, in this subsection, we evaluate the time cost of Arranging in comparison with Perfect GC, which reflects the overhead for updating the OPT data layout in MFDedup.

GC approaches mainly differ in the technique to select the containers and chunks to clean. Once selected though, all the GC techniques involve migrating referenced chunks into new, immutable containers. To simplify our evaluation, we conservatively focus on the cost of reading selected containers and migrating valid chunks into new containers since that is the common phase. This is a lower bound on the cost of GC since it neglects the selection phase, which involves enumerating the live files/chunks [11].

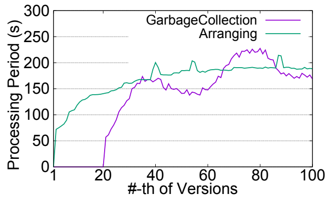
The results are shown in Figure 12 comparing Arranging and Perfect GC. Since we are retaining 20 versions, GC does not run for the first 20 versions, though Arranging does. Analyzing the steady-state performance after the 20th version, Arranging's total processing period is only 45% (WEB), 37% (CHM) and 25% (SYN) of GC's total processing time on average. But in VMS, Arranging takes 9% longer than GC because VMS's modification style (always change the same region in each backup) makes GC very easy. Generally, Figure 12 suggests that Arranging is usually faster than GC, which would take even more time if the selection phase were included in GC's total. When MFDedup runs its version of GC,



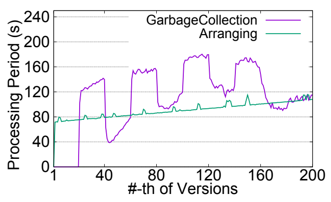
(a) WEB



(b) CHM

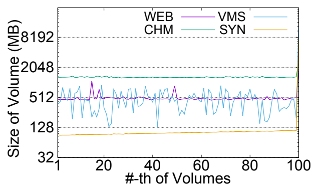


(c) VMS

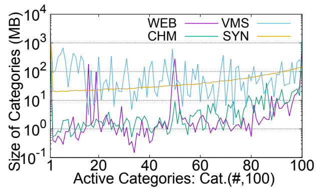


(d) SYN

Figure 12: Time cost comparison between Arranging of MFDedup and GC of traditional deduplication systems.



(a) Volume



(b) Active Category

Figure 13: Size distribution of Volumes and Categories after deduplicating 100 backup versions with MFDedup.

the processing time is insignificant since large Volumes can be deleted at once without any copy-forward.

Arranging has a consistent processing time across versions, while GC's runtime is more variable, and consistent overheads are easier to plan for in a storage system. Arranging's processing time is consistent because it is a local process on a recent version, while GC is a global process. As Figure 6 shows, Arranging is always applied in Active categories generated in the same version, and it always achieves a better locality. On the other hand, GC in other techniques suffers from poor locality [17], because the selected containers and chunks are distributed randomly.

Note that other GC approaches will be faster than Perfect GC, but at the cost of greatly decreasing Actual Deduplication Ratio as discussed in Section 5.2. In contrast, MFDedup has almost no deduplication ratio loss for GC while supporting immediate deletion and GC, and also achieving nearly perfect restore performance with an acceptable Arranging cost, as shown in Figures 7, 9, and 12.

5.6 Size Distribution of Volumes/Categories

In this section, we demonstrate the data layout of MFDedup with the size of volumes and active categories. We back up 100 versions without retention for evaluation. After that, there will be 99 Volumes and 100 active categories, and these active categories compose a logical volume (they will be archived in a volume after the next Arranging).

Figure 13(a) shows the size of Volumes varying from 90MB to 1.3GB on our four datasets. The results can tell administrators how much space will be freed by MFDedup by deleting backup versions (volumes). The results also help administrators estimate how much more can be written to the deduplication system, since volumes represent the difference between neighboring versions. For previous deduplication systems, it is difficult to answer these two issues [35], though sketching approaches have been considered [19].

Figure 13(b) shows the size of active Categories vary in a large range. We learn that the maximum categories hold about 16.99% (WEB), 46.46% (CHM), 18.49% (VMS), 51.87% (SYN) of the size of the last backup version. This indicates the reserved-space requirement for offline Arranging in MFDedup is much smaller than a full backup as discussed in Section 4.5. Meanwhile, the reserved space can be further reduced by compressing categories.

6 Conclusion and Future Work

In this paper, we propose a management-friendly deduplication framework, MFDedup. Different from traditional 'Write Friendly' style deduplication architectures, MFDedup, is designed to be 'Management-Friendly' and solves the fragmentation problem in deduplication-based backup systems, by introducing a novel deduplication process (NDF) and a locality improvement process (AVAR) to generate OPT data layout and thus maintain locality of backup workloads.

With the benefits of eliminating the fragmentation problem, MFDedup improves actual deduplication ratios (1.12× to 2.19× higher) and restore throughput (2.63× to 11.64× higher) than previous approaches with accepted time cost on the offline 'Arranging' to update the OPT data layout, while GC in MFDedup is nearly zero-overhead.

As future work, we are considering adding delta compression in MFDedup for further space savings as well as handling more complex backup scenarios such as incremental backups.

Acknowledgments

We are grateful to our shepherd Danny Harnik and the anonymous reviewers for their insightful comments. This research was partly supported by National Key R&D Program of China under Grant no. 2018YFB1003800, 2018YFB1003805, the National Natural Science Foundation of China under Grant no. 61972441, no. 61972112, and no. 61832004, the Shenzhen Science and Technology Program under Grant no. JCYJ20190806143405318, no. JCYJ20200109113427092, and no. JCYJ20170413105929681, Innovation Fund of WNLO 2018WNLOKF008.

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] Intel intelligent storage acceleration library crypto version. https://github.com/intel/isa-l_crypto. [Online].

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

[3] George Amvrosiadis and Medha Bhadkamkar. Identifying trends in enterprise data protection systems. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC '15)*, page 151–164, USA, September 2015. USENIX Association.

[4] George Amvrosiadis and Medha Bhadkamkar. Getting back up: Understanding how enterprise data backups fail. In *Proceedings of the 2016 USENIX Conference on USENIX Annual Technical Conference (ATC' 16)*, pages 479–492, Denver, CO, June 2016. USENIX Association.

[5] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T Klein. The design of a similarity based deduplication system. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–14, Haifa, Israel, October 2009. Association for Computing Machinery.

[6] Tony Asaro and Heidi Biggar. Data de-duplication and disk-to-disk backup systems: Technical and business considerations. *The Enterprise Strategy Group*, pages 2–15, 2007.

[7] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 1–9. IEEE, 2009.

[8] William J Bolosky, Scott Corbin, David Goebel, and John R Douceur. Single instance storage in windows 2000. In *Proceedings of the 4th USENIX Windows Systems Symposium*, pages 13–24, Seattle, WA, August 2000. USENIX Association.

[9] 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 (FAST' 19)*, pages 129–142, Boston, MA, February 2019. USENIX Association.

[10] Zhichao Cao, Hao Wen, Fenggang Wu, and David H. C. Du. Alacc: Accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST' 18)*, pages 309–324, Oakland, CA, USA, February 2018. USENIX Association.

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

[12] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication—large scale study and system design. In *Presented as part of the 2012 USENIX Annual Technical Conference (ATC'12)*, pages 285–296, Boston, MA, October 2012. USENIX Association.

[13] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30(2005), 2005.

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

[15] 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 (ATC'14)*, page 181–192, Philadelphia, PA, October 2014. USENIX Association.

[16] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*, page 331–344, Santa Clara, CA, February 2015. USENIX Association.

[17] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC' 11)*, pages 1–25, Portland, OR, October 2011. USENIX Association.

[18] Sangwook Shane Hahn, Sungjin Lee, Cheng Ji, Li-Pin Chang, Inhyuk Yee, Liang Shi, Chun Jason Xue, and Jihong Kim. Improving file system performance of mobile storage systems using a decoupled defragmenter. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (ATC' 17)*, pages 759–771, Santa Clara, CA, July 2017. USENIX Association.

[19] Danny Harnik, Moshik Hershcovitch, Yosef Shatsky, Amir Epstein, and Ronen Kat. Sketching volume capacities in deduplicated storage. *ACM Transactions on Storage*, 15(4), December 2019.

[20] Muhammad Asim Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. mos: A reusable networking stack for flow monitoring middleboxes. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 113–129, Boston, MA, March 2017.

[21] Keren Jin and Ethan L Miller. The effectiveness of deduplication on virtual machine disk images. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–12, Haifa, Israel, oct 2009. Association for Computing Machinery.

[22] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR '12, Haifa, Israel, October 2012. Association for Computing Machinery.

[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 (FAST'13)*, page 183–198, San Jose, CA, February 2013. USENIX Association.

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

[25] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. Read-performance optimization for deduplication-based storage systems in the cloud. *ACM Trans. Storage*, 10(2), March 2014.

[26] Dirk Meister, Jürgen Kaiser, and André Brinkmann. Block locality caching for data deduplication. In *Proceedings of the 6th International Systems and Storage Conference (SYSTOR'13)*, Haifa, Israel, October 2013. Association for Computing Machinery.

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

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

[29] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP' 01)*, pages 174–187, Banff, Alberta, Canada, December 2001. Association for Computing Machinery.

[30] Young Jin Nam, Dongchul Park, and David H.C. Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '12)*, pages 201–208, USA, July 2012. IEEE Computer Society.

[31] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proceedings of the 2011 IEEE International Conference on High Performance Computing and Communications (HPCC' 11)*, pages 581–586, USA, July 2011. IEEE, IEEE Computer Society.

[32] Fan Ni and Song Jiang. Rapidcdc: Leveraging duplicate locality to accelerate chunking in cdc-based deduplication systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC' 19)*, pages 220–232, Santa Cruz, CA, USA, November 2019. Association for Computing Machinery.

[33] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC'04)*, pages 73–86, Boston, MA, October 2004. USENIX Association.

[34] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02*, volume 2, pages 89–101, Monterey, CA, February 2002. USENIX Association.

[35] Philip Shilane, Ravi Chitloor, and Uday Kiran Jonnala. 99 deduplication problems. In *Proceedings of the 8th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'16)*, page 86–90, Denver, CO, USA, June 2016. USENIX Association.

[36] 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 (ATC'12)*, pages 261–272, Boston, MA, 2012. USENIX Association.

[37] Michael Vrable, Stefan Savage, and Geoffrey M Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage (TOS)*, 5(4):1–28, 2009.

[38] 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 (FAST'12)*, volume 12, pages 4–4, San Jose, CA, February 2012. USENIX Association.

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

[40] Wen Xia, Hong Jiang, Dan Feng, and Yu Hua. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference (ATC' 11)*, pages 26–30, Portland, OR, October 2011. USENIX Association.

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

[42] Yucheng Zhang, Hong Jiang, Dan Feng, Wen Xia, Min Fu, Fangting Huang, and Yukun Zhou. Ae: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. In *2015 IEEE Conference on Computer Communications (INFOCOM)*, pages 1337–1345. IEEE, August 2015.

[43] 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 17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 121–128, Santa Clara, CA, USA, February 2019. USENIX Association.

[44] Nannan Zhao, Hadeel Albahar, Subil Abraham, Keren Chen, Vasily Tarasov, Dimitrios Skourtis, Lukas Rupprecht, Ali Anwar, and Ali R. Butt. Duphunter: Flexible high-performance deduplication for docker registries. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 769–783. USENIX Association, July 2020.

[45] 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 (FAST' 08)*, volume 8, pages 1–14, San Jose, California, February 2008. USENIX Association.