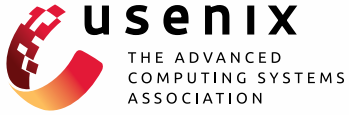

Design Tradeoffs for Data Deduplication Performance in Backup Workloads

Min Fu, Dan Feng, and Yu Hua, *Huazhong University of Science and Technology;*
Xubin He, *Virginia Commonwealth University;* **Zuoning Chen,** *National Engineering Research Center for Parallel Computer;* **Wen Xia and Yucheng Zhang,** *Huazhong University of Science and Technology;* **Yujuan Tan,** *Chongqing University*

https://www.usenix.org/conference/fast15/technical-sessions/presentation/fu

This paper is included in the Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15).

February 16–19, 2015 • Santa Clara, CA, USA

ISBN 978-1-931971-201

Open access to the Proceedings of the 13th USENIX Conference on File and Storage Technologies is sponsored by USENIX

Design Tradeoffs for Data Deduplication Performance in Backup Workloads

Min Fu[†], Dan Feng[†], Yu Hua[†], Xubin He[‡], Zuoning Chen[*], Wen Xia[†], Yucheng Zhang[†], Yujuan Tan[§]
[†]*Wuhan National Lab for Optoelectronics*
School of Computer, Huazhong University of Science and Technology, Wuhan, China
[‡]*Dept. of Electrical and Computer Engineering, Virginia Commonwealth University, VA, USA*
[*]*National Engineering Research Center for Parallel Computer, Beijing, China*
[§]*College of Computer Science, Chongqing University, Chongqing, China*
Corresponding author: dfeng@hust.edu.cn

Abstract

Data deduplication has become a standard component in modern backup systems. In order to understand the fundamental tradeoffs in each of its design choices (such as prefetching and sampling), we disassemble data deduplication into a large N-dimensional parameter space. Each point in the space is of various parameter settings, and performs a tradeoff among backup and restore performance, memory footprint, and storage cost. Existing and potential solutions can be considered as specific points in the space. Then, we propose a general-purpose framework to evaluate various deduplication solutions in the space. Given that no single solution is perfect in all metrics, our goal is to find some reasonable solutions that have sustained backup performance and perform a suitable tradeoff between deduplication ratio, memory footprints, and restore performance. Our findings from extensive experiments using real-world workloads provide a detailed guide to make efficient design decisions according to the desired tradeoff.

1 Introduction

Efficient storage of a huge volume of digital data becomes a big challenge for industry and academia. IDC predicts that there will be 44 ZB of digital data in 2020 [3], which will continue to grow exponentially. Recent work reveals the wide existence of a large amount of duplicate data in storage systems, including primary storage systems [27, 37], secondary backup systems [36], and high-performance data centers [26]. In order to support efficient storage, data deduplication is a widely deployed technique between the underlying storage system and upper applications due to its efficiency and scalability; it becomes increasingly important in large-scale storage systems, especially backup systems.

In backup systems, data deduplication divides a backup stream into non-overlapping variable-sized chunks, and identifies each chunk with a cryptographic digest, such as SHA-1, commonly referred to as a *fingerprint*. Two chunks with identical fingerprints are considered duplicates without requiring a byte-by-byte comparison. The probability of hash collisions is much smaller than that of hardware errors [33], thus it is widely accepted in real-world backup systems. A *fingerprint index* maps fingerprints of the stored chunks to their physical addresses. A duplicate chunk can be identified via checking the existence of its fingerprint in the index. During a backup, the duplicate chunks are eliminated immediately for inline data deduplication. The chunks with unique fingerprints that do not exist in the fingerprint index are aggregated into fixed-sized *containers* (typically 4 MB), which are managed in a log-structure manner [39]. A *recipe* that consists of the fingerprint sequence of the backup is written for future data recovery.

There have been many publications about data deduplication [32]. However, it remains unclear how existing solutions make their design decisions and whether potential solutions can do better. Hence, in the first part of the paper (Section 2), we present a taxonomy to classify existing work using individual design parameters, including **key-value**, **fingerprint prefetching and caching**, **segmenting**, **sampling**, **rewriting**, **restore**, etc. Different from previous surveys [24, 32], our taxonomy is fine-grained with in-depth discussions. We obtain an N-dimensional parameter space, and each point in the space performs a tradeoff among backup and restore performance, memory footprint, and storage cost. Existing solutions are considered as specific points. We figure out how existing solutions choose their points, which allows us to find potentially better solutions. For example, similarity detection in Sparse Indexing [22] and segment prefetching in SiLo [38] are highly complementary.

Although there are some open-source deduplication platforms, such as dmdedup [35], none of them are capable of evaluating the parameter space we discuss. Hence, the second part of our paper (Section 3)

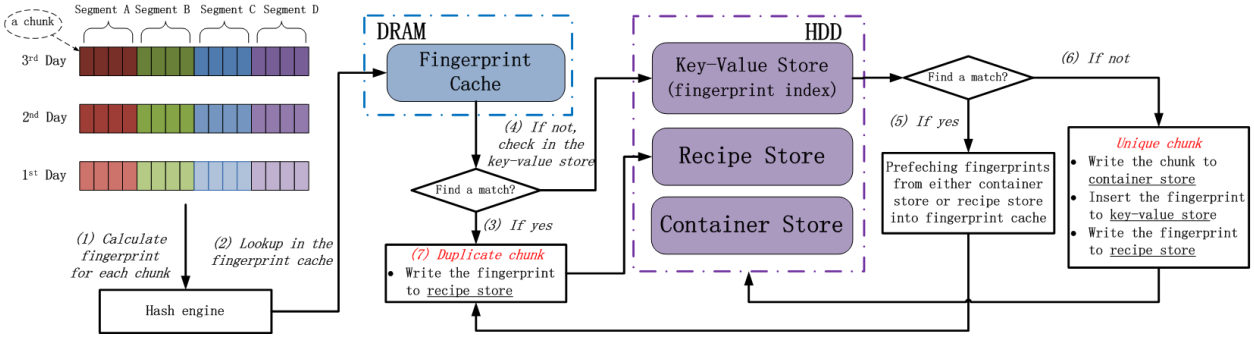

Figure 1: A typical deduplication system and the **Base** deduplication procedure.

presents a general-purpose Deduplication Framework (**DeFrame**)[1], for comprehensive data deduplication evaluation. DeFrame implements the entire parameter space discussed in a modular and extensible fashion; this enables apple-to-apple comparisons among both existing and potential solutions. Our aim is to facilitate finding solutions that provide sustained high backup and restore performance, low memory footprints, and high storage efficiency.

The third part of our paper (Section 4) presents our findings in a large-scale experimental evaluation using real-world long-term workloads. The findings provide a detailed guide to make reasonable decisions according to the desired tradeoff. For example, if high restore performance is required, a rewriting algorithm is required to trade storage efficiency for restore performance. With a rewriting algorithm, the design decisions on the fingerprint index need to be changed. To the best of our knowledge, this is the first work that examines the interplays between fingerprint index and rewriting algorithms.

2 In-line Data Deduplication Space

Figure 1 depicts a typical deduplication system. Generally, we have three components on disks: (1) The *fingerprint index* maps fingerprints of stored chunks to their physical locations. It is used to identify duplicate chunks. (2) The *recipe store* manages recipes that describe the logical fingerprint sequences of concluded backups. A recipe is used to reconstruct a backup stream during restore. (3) The *container store* is a log-structured storage system. While duplicate chunks are eliminated, unique chunks are aggregated into fixed-sized containers. We also have a *fingerprint cache* in DRAM that holds popular fingerprints to boost duplication identification.

Figure 1 also shows the basic deduplication procedure, namely *Base*. At the top left, we have three sample backup streams that correspond to the snapshots of the primary data in three consecutive days. Each backup stream is divided into chunks, and 4 consecutive chunks constitute a *segment* (a segment describes the chunk sequence of a piece of data stream; we assume a simplest segmenting approach the this case). Each chunk is processed in the following steps: **(1)** The hash engine calculates the SHA-1 digest for the chunk as its unique identification, namely fingerprint. **(2)** Look up the fingerprint in the in-DRAM fingerprint cache. **(3)** If we find a match, jump to step 7. **(4)** Otherwise, look up the fingerprint in the key-value store. **(5)** If we find a match, invoke a fingerprint prefetching procedure. Jump to step 7. **(6)** Otherwise, the chunk is unique. We write the chunk to the container store, insert the fingerprint to the key-value store, and write the fingerprint to the recipe store. Jump to step 1 to process the next chunk. **(7)** The chunk is a duplicate. We eliminate the chunk and write its fingerprint to the recipe store. Jump to step 1 to process the next chunk.

Parameter list	Description
sampling	selecting representative fingerprints
segmenting	splitting the unit of logical locality
segment selection	selecting segments to be prefetched
segment prefetching	exploiting segment-level locality
key-value mapping	multiple logical positions per fingerprint
rewriting algorithm	reducing fragmentation
restore algorithm	designing restore cache

Table 1: The major parameters we discuss.

In the following sections, we (1) propose the fingerprint index subspace (the key component in data deduplication systems) to characterize existing solutions and find potentially better solutions, and (2) discuss the interplays among fingerprint index, rewriting, and restore algorithms. Table 1 lists the major parameters.

2.1 Fingerprint Index

The fingerprint index is a well-recognized performance bottleneck in large-scale deduplication systems [39]. The simplest fingerprint index is only a key-value store [33]. The key is a fingerprint and the value points to the chunk. A duplicate chunk is identified via checking the existence of its fingerprint in the key-value store. Suppose each key-value pair consumes 32 bytes (including a 20-byte fingerprint, an 8-byte container ID, and 4-byte other metadata) and the chunk size is 4 KB on average, indexing 1 TB unique data requires at least an 8 GB-sized key-value store. Putting all fingerprints in DRAM is not

[1] https://github.com/fomy/destor

cost-efficient. To model the storage cost, we use the unit price from Amazon.com [1]: A Western Digital Blue 1 TB 7200 RPM SATA Hard Drive costs $60, and a Kingston HyperX Blu 8 GB 1600 MHz DDR3 DRAM costs $80. The total storage cost is $140, 57.14% of which is for DRAM.

An HDD-based key-value store suffers from HDD's poor random-access performance, since the fingerprint is completely random in nature. For example, the throughput of Content-Defined Chunking (CDC) is about 400 MB/s under commercial CPUs [11], and hence CDC produces 102,400 chunks per second. Each chunk incurs a lookup request to the key-value store, i.e., 102,400 lookup requests per second. The required throughput is significantly higher than that of HDDs, i.e., 100 IOPS [14]. SSDs support much higher throughput, nearly 75,000 IOPS as venders report [4]. However, SSDs are much more expensive than HDDs and suffer from a performance degradation over time due to reduced over-provisioning space [19].

Due to the incremental nature of backup workloads, the fingerprints of consecutive backups appear in similar sequences [39], which is known as *locality*. In order to reduce the overhead of the key-value store, modern fingerprint indexes leverage locality to prefetch fingerprints, and maintain a *fingerprint cache* to hold the prefetched fingerprints in memory. The fingerprint index hence consists of two submodules: a key-value store and a fingerprint prefetching/caching module. The value instead points to the prefetching unit. According to the use of the key-value store, we classify the fingerprint index into *exact* and *near-exact deduplication*.

- **Exact Deduplication (ED)**: all duplicate chunks are eliminated for highest deduplication ratio (the data size before deduplication divided by the data size after deduplication).
- **Near-exact Deduplication (ND)**: a small number of duplicate chunks are allowed for higher backup performance and lower memory footprint.

According to the fingerprint prefetching policy, we classify the fingerprint index into exploiting *logical* and *physical locality*.

- **Logical Locality (LL)**: the chunk (fingerprint) sequence of a backup stream before deduplication. It is preserved in recipes.
- **Physical Locality (PL)**: the physical layout of chunks (fingerprints), namely the chunk sequence after deduplication. It is preserved in containers.

Figure 2 shows the categories of existing fingerprint indexes. The cross-product of the deduplication and locality variations include **EDPL**, **EDLL**, **NDPL**, and **NDLL**. In the following, we discuss their parameter subspaces and how to choose reasonable parameter settings.

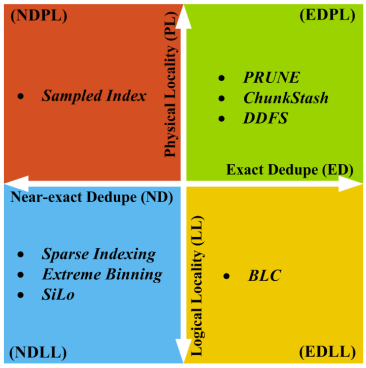

Figure 2: Categories of existing fingerprint indexes. The typical examples include DDFS [39], Sparse Indexing [22], Extreme Binning [10], ChunkStash [14], Sampled Index [18], SiLo [38], PRUNE [28], and BLC [25].

2.1.1 Exact vs. Near-exact Deduplication

The main difference between exact and near-exact deduplication is the use of the key-value store. For exact deduplication, the key-value store has to index the fingerprints of all stored chunks and hence becomes too large to be stored in DRAM. *The fingerprint prefetching/caching module is employed to avoid a large fraction of lookup requests to the key-value store.* Due to the strong locality in backup workloads, the prefetched fingerprints are possibly accessed later. Although the fingerprint index is typically lookup-intensive (most chunks are duplicate in backup workloads), its key-value store is expected not to be lookup-intensive since a large fraction of lookup requests are avoided by the fingerprint prefetching and caching. However, the fragmentation problem discussed in Section 2.1.2 reduces the efficiency of the fingerprint prefetching and caching, making the key-value store become lookup-intensive over time.

For near-exact deduplication, only sampled representative fingerprints, namely *features*, are indexed to downsize the key-value store. With a high sampling ratio (e.g., 128:1), the key-value store is small enough to be completely stored in DRAM. Since only a small fraction of stored chunks are indexed, many duplicate chunks cannot be found in the key-value store. *The fingerprint prefetching/caching module is important to maintain a high deduplication ratio.* Once an indexed duplicate fingerprint is found, many unindexed fingerprints are prefetched to answer following lookup requests. The sampling method is important to the prefetching efficiency, and hence needs to be chosen carefully, which are discussed in Section 2.1.2 and 2.1.3 respectively.

The memory footprint of exact deduplication is related to the key-value store, and proportional to the number

of the stored chunks. For example, if we employ Berkeley DB [2] paired with a Bloom filter [12] as the key-value store, 1 byte DRAM per stored chunk is required to maintain a low false positive ratio for the Bloom filter. Suppose S is the average chunk size in KB and M is the DRAM bytes per key, the storage cost per TB stored data includes $\$(\frac{10*M}{S})$ for DRAM and \$60 for HDD. Given $M = 1$ and $S = 4$, the storage cost per TB stored data is \$62.5 (4% for DRAM). In other underlying storage, such as RAID [31], the proportion of DRAM decreases.

The memory footprint of near-exact deduplication depends on the sampling ratio R. Suppose each key-value pair consumes 32 bytes, M is equal to $\frac{32}{R}$. The storage cost per TB stored data includes $\$(\frac{320}{S*R})$ for DRAM and \$60 for HDD. For example, if $R = 128$ and $S = 4$, the storage cost per TB data in near-exact deduplication is \$60.625 (1% for DRAM). However, it is unfair to simply claim that near-exact deduplication saves money, since its stored data includes duplicate chunks. Suppose a 10% loss of deduplication ratio, 1 TB data in near-exact deduplication only stores 90% of 1 TB data in exact deduplication. Hence, near-exact deduplication requires 1.11 TB to store the 1 TB data in exact deduplication. The total storage cost in near-exact deduplication is about \$67.36, higher than exact deduplication. To decrease storage cost, near-exact deduplication has to achieve a high deduplication ratio, no smaller than $(\frac{320}{S*R} + 60)/(\frac{10}{S} + 60)$ of the deduplication ratio in exact deduplication. In our cost model, near-exact deduplication needs to achieve 97% of the deduplication ratio of exact deduplication, which is difficult based on our observations in Section 4.7. Hence, near-exact deduplication generally indicates a cost increase.

2.1.2 Exploiting Physical Locality

The unique chunks (fingerprints) of a backup are aggregated into containers. Due to the incremental nature of backup workloads, the fingerprints in a container are possibly accessed together in subsequent backups [39]. The locality preserved in containers is called *physical locality*. To exploit the physical locality, the value in the key-value store is the container ID and thus the prefetching unit is a container. If a duplicate fingerprint is identified in the key-value store, we obtain a container ID and then read the metadata section (a summary on the fingerprints) of the container into the fingerprint cache. Note that only unique fingerprints are updated with their container IDs in the key-value store.

Although physical locality is an effective approximation of logical locality, the deviation increases over time. For example, old containers have many useless fingerprints for new backups. As a result, the efficiency of the fingerprint prefetching/caching module decreases over time. This problem is known as *fragmentation*, which severely decreases restore performance as reported in recent work [29, 21]. For EDPL, the fragmentation gradually changes the key-value store to be lookup-intensive, and the ever-increasing lookup overhead results in unpredictable backup performance. We cannot know when the fingerprint index will become the performance bottleneck in an aged system.

For NDPL, the sampling method has significant impacts on deduplication ratio. We observe two sampling methods, *uniform* and *random*. The former selects the first fingerprint every R fingerprints in a container, while the latter selects the fingerprints that $mod\ R = 0$ in a container. Although Sampled Index [18] uses the random sampling, we observe the uniform sampling is better. In the random sampling, the missed duplicate fingerprints would not be sampled ($mod\ R \neq 0$) after being written to new containers, making new containers have less features and hence smaller probability of being prefetched. Without this problem, the uniform sampling achieves a significantly higher deduplication ratio.

2.1.3 Exploiting Logical Locality

To exploit logical locality preserved in recipes, each recipe is divided into subsequences called *segments*. A segment describes a fingerprint subsequence of a backup, and maps its fingerprints to container IDs. We identify each segment by a unique ID. The value in the key-value store points to a segment instead of a container, and the segment becomes the prefetching unit. Due to the locality preserved in the segment, the prefetched fingerprints are possibly accessed later. Note that in addition to unique fingerprints, duplicate fingerprints have new segment IDs (unlike physical locality).

For exact deduplication whose key-value store is not in DRAM, it is necessary to access the key-value store as infrequently as possible. Since the Base procedure depicted in Figure 1 follows this principle (only missed-in-cache fingerprints are checked in the key-value store), it is suitable for EDLL. A problem in EDLL is frequently updating the key-value store, since unique and duplicate fingerprints both have new segment IDs. As a result, all fingerprints are updated with their new segment IDs in the key-value store. The extremely high update overhead, which has not been discussed in previous studies, either rapidly wears out an SSD-based key-value store or exhausts the HDD bandwidth. We propose to sample features in segments and only update the segment IDs of unique fingerprints and features in the key-value store. In theory, the sampling would increase lookup overhead, since it leaves many fingerprints along with old segment IDs, leading to a suboptimal prefetching efficiency. However, based on our observations in Section 4.3, the increase of lookup overhead is negligible, making it a reasonable tradeoff. One problem of the sampling optimization is that, after users delete some backups, the

	BLC [25]	Extreme Binning [10]	Sparse Index [22]	SiLo [38]
Exact deduplication	Yes	No	No	No
Segmenting method	FSS	FDS	CDS	FSS & FDS
Sampling method	N/A	Minimum	Random	Minimum
Segment selection	Base	Top-*all*	Top-*k*	Top-1
Segment prefetching	Yes	No	No	Yes
Key-value mapping relationship	1:1	1:1	Varied	1:1

Table 2: Design choices for exploiting logical locality.

fingerprints pointing to stale segments may become unreachable. It would decrease deduplication ratio. A possible solution is to add a column in key-value store to keep container IDs. Only the fingerprints in reclaimed containers are removed in key-value store. The additional storage cost is negligible since EDLL keeps key-value store on disks.

Previous studies on NDLL use similarity detection (described later) instead of the Base procedure [22, 10, 38]. Given the more complicated logic frame and additional in-memory buffer in similarity detection, it is still necessary to make the motivation clear. Based on our observations in Section 4.5, the Base procedure performs well in source code and database datasets (datasets are described in Section 4.1), but underperforms in virtual machine dataset. The major characteristic of virtual machine images is that each image itself contains many duplicate chunks, namely *self-reference*. Self-reference, absent in our source code and database datasets, interferes with the prefetching decision in the Base procedure. A more efficient fingerprint prefetching policy is hence desired in complicated datasets like virtual machine images. As a solution, similarity detection uses a buffer to hold the processing segment, and load the most similar stored segments in order to deduplicate the processing segment. We summarize existing fingerprint indexes exploiting logical locality in Table 2, and discuss similarity detection in a 5-dimensional parameter subspace: segmenting, sampling, segment selection, segment prefetching, and key-value mapping. Note that the following methods of segmenting, sampling, and prefetching are also applicable in the Base procedure.

Segmenting method. The File-Defined Segmenting (FDS) considers each file as a segment [10], which suffers from the greatly varied file size. The Fixed-Sized Segmenting (FSS) aggregates a fixed number (or size) of chunks into a segment [38]. FSS suffers from a shifted content problem similar to the Fixed-Sized Chunking method, since a single chunk insertion/deletion completely changes the segment boundaries. The Content-Defined Segmenting method (CDS) checks the fingerprints in the backup stream [22, 16]. If a chunk's fingerprint matches some predefined rules (e.g., last 10 bits are zeros), the chunk is considered as a segment boundary. CDS is shift-resistant.

Sampling method. It is impractical to calculate the exact similarity of two segments using their all fingerprints. According to Broder [13], the similarity of the two randomly sampled subsets is an unbiased approximation of that of the two complete sets. A segment is considered as a set of fingerprints. A subset of the fingerprints are selected as features since the fingerprints are already random. If two segments share some features, they are considered similar. There are three basic sampling methods: *uniform*, *random*, and *minimum*. The uniform and random sampling methods have been explained in Section 2.1.2. Suppose the sampling ratio is R, the minimum sampling selects the $\frac{segment\ length}{R}$ minimum fingerprints in a segment. Since the distribution of minimum fingerprints is uneven, Aronovich et al. propose to select the fingerprint adjacent to the minimum fingerprint [9]. Only sampled fingerprints are indexed in the key-value store. A smaller R provides more candidates for the segment selection at a cost of increasing the memory footprint. One feature per segment forces a single candidate. The uniform sampling suffers from the problem of content shifting, while the random and minimum sampling are shift-resistant.

Segment selection. After features are sampled in a new segment S, we look up the features in the key-value store to find the IDs of similar segments. There may be many candidates, but not all of them are loaded in the fingerprint cache since too many segment reads hurt backup performance. The similarity-based selection, namely Top-k, selects k most similar segments. Its procedure is as follows: (1) a most similar segment that shares most features with S is selected at a time; (2) the features of the selected segment are eliminated in remaining candidates, to avoid giving scores for features belonging to already selected segments; (3) jump to step 1 to select the next similar segment, until k of segments are selected or we run out of candidates [22]. A more aggressive selection method is to read all similar segments together, namely Top-*all*. A necessary optimization for Top-*all* is to physically aggregate similar segments into a *bin*, and thus a single I/O can fetch all similar segments [10]. A dedicated bin store is hence required. It underperforms if we sample more than 1 feature per segment, since the bins grow big quickly.

Figure 1 illustrates how similar segments arise. Suppose A_1 is the first version of segment A, A_2 is the second version, and so on. Due to the incremental nature

of backup workloads, A_3 is possibly similar to its earlier versions, i.e., A_1 and A_2. Such kind of similar segments are *time-oriented*. Generally, reading only the latest version is sufficient. An exceptional case is a segment boundary change, which frequently occurs if fixed-sized segmenting is used. A boundary change may move a part of segment B to segment A, and hence A_3 has two time-oriented similar segments, A_2 and B_2. A larger k is desired to handle these situations. In datasets like virtual machine images, A can be similar with other segments, such as E, due to self-reference. These similar segments are *space-oriented*. Suppose A_2 and E_2 both have 8 features, 6 of them are shared. After deduplicating E_2 at a later time than A_2, 6 of A_2's features are overwritten to be mapped to E_2. E_2 is selected prior to A_2 when deduplicating A_3. A larger k increases the probability of reading A_2. Hence, with many space-oriented segments, a larger k is required to accurately read the time-oriented similar segment at a cost of decreased backup performance.

Segment prefetching. SiLo exploits segment-level locality to prefetch segments [38]. Suppose A_3 is similar to A_2, it is reasonable to expect the next segment B_3 is similar to B_2 that is next to A_2. Fortunately, B_2 is adjacent to A_2 in the recipe, and hence $p-1$ segments following A_2 are prefetched when deduplicating A_3. If more than 1 similar segments are read, segment prefetching can be applied to either all similar segments, as long as $k*p$ segments do not overflow the fingerprint cache, or only the most similar segment.

Segment prefetching has at least two advantages: 1) *Reducing lookup overhead*. Prefetching B_2 together with A_2, while deduplicating A_3, avoids the additional I/O of reading B_2 for the following B_3. 2) *Improving deduplication ratio*. Assuming the similarity detection fails for B_3 (i.e., B_2 is not found due to its features being changed), the previous segment prefetching caused by A_3, whose similarity detection succeeds (i.e., A_2 is read for A_3 and B_2 is prefetched together), offers a deduplication opportunity for B_3. Segment prefetching also alleviates the problem caused by segment boundary change. In the case of two time-oriented similar segments, A_2 and B_2 would be prefetched for A_3 even if $k=1$. Segment prefetching relies on storing segments in a logical sequence being incompatible with Top-*all*.

Key-value mapping relationship. The key-value store maps features to stored segments. Since a feature can belong to different segments (hence multiple logical positions), the key can be mapped to multiple segment IDs. As a result, the value becomes a FIFO queue of segment IDs, where v is the queue size. For NDLL, maintaining the queues has low performance overhead since the key-value store is in DRAM. A larger v provides more similar segment candidates at a cost of a higher memory footprint. It is useful in the following cases: 1) *Self-reference is common*. A larger v alleviates the above problem of feature overwrites caused by space-oriented similar segments. 2) *The corrupted primary data is restored to an earlier version rather than the latest version (rollback)*. For example, if A_2 has some errors, we roll back to A_1 and thus A_3 derives from A_1 rather than A_2. In this case, A_1 is a better candidate than A_2 for deduplicating A_3, however features of A_1 have been overwritten by A_2. A larger v avoids this problem.

2.2 Rewriting and Restore Algorithms

Since the fragmentation decreases the restore performance in aged systems, the rewriting algorithm, an emerging dimension in the parameter space, was proposed to allow sustained high restore performance [20, 21, 17]. It identifies fragmented duplicate chunks and rewrites them to new containers. Even though near-exact deduplication trades deduplication ratio for restore performance, our observations in Section 4.6 show that the rewriting algorithm is a more efficient tradeoff. However, the rewriting algorithm's interplay with the fingerprint index has not yet been discussed.

We are mainly concerned about two questions. (1) *How does the rewriting algorithm reduce the ever-increasing lookup overhead of EDPL?* Since the rewriting algorithm reduces the fragmentation, EDPL is improved because of better physical locality. Our observations in Section 4.6 show that, via an efficient rewriting algorithm, the lookup overhead of EDPL no longer increases over time. EDPL then has sustained backup performance. (2) *Does the fingerprint index return the latest container ID when a recently rewritten chunk is checked?* Each rewritten chunk would have a new container ID. If the old container ID is returned when that chunk is recirculated, then another rewrite could occur. Based on our observations, this problem is more pronounced and significant in EDLL than EDPL. An intuitive explanation is that, due to our sampling optimization mentioned in Section 2.1.3, an old segment containing obsolete container IDs is read for deduplication. As a result, EDPL becomes better than EDLL due to its higher deduplication ratio.

While the rewriting algorithm determines the chunk placement, an efficient restore algorithm leverages the placement to gain better restore performance with a limited memory footprint. There have been three restore algorithms: the basic LRU cache, the forward assembly area (ASM) [21], and the optimal cache (OPT) [17]. In all of them, a container serves as the prefetching unit during a restore to leverage locality. Their major difference is that while LRU and OPT use container-level replacement, ASM uses chunk-level replacement. We observe these algorithms' performances under different placements in Section 4.6. If the fragmentation is dominant, ASM is more efficient. The reason is that LRU

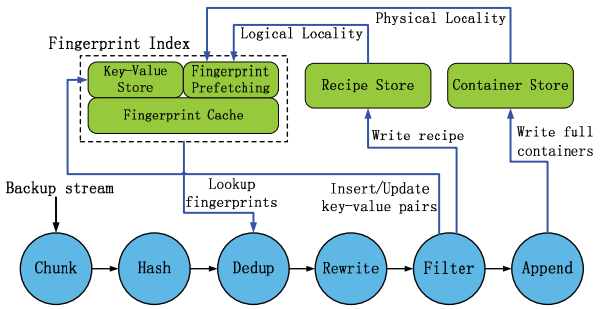

Figure 3: The architecture and backup pipeline.

and OPT hold many useless chunks that are not restored in DRAM due to their container-level replacement. On the other hand, if an efficient rewriting algorithm has reduced the fragmentation, the container-level replacement is improved due to better physical locality and OPT performs best due to its accurate cache replacement.

3 The DeFrame Framework

The N-dimensional parameter space discussed in Section 2 provides a large amount of design choices, but there is no platform to evaluate these choices as far as we know. In this section, we present *DeFrame* as a general-purpose chunk-level deduplication framework to facilitate exploring alternatives. In DeFrame, existing and potential solutions are considered as specific points in the N-dimensional parameter space. We implement DeFrame using C and pthreads on 64-bit Linux.

3.1 Architecture

As shown in Figure 3, DeFrame consists of three submodules discussed in Section 2. In the *container store*, each container is identified by a globally unique ID. The container is the prefetching unit for exploiting physical locality. Each container includes a metadata section that summarizes the fingerprints of all chunks in the container. We can fetch an entire container or only its metadata section via a container ID.

The *recipe store* manages recipes of all finished backups. In recipes, the associated container IDs are stored along with fingerprints so as to restore a backup without the need to consult the fingerprint index. We add some indicators of segment boundaries in each recipe to facilitate reading a segment that is the prefetching unit for exploiting logical locality. Each segment is identified by a globally unique ID. For example, an ID can consist of a 2-byte pointer to a recipe, a 4-byte offset in the recipe, and a 2-byte segment size that indicates how many chunks are in the segment.

The *fingerprint index* consists of a key-value store and a fingerprint prefetching/caching module. Two kinds of key-value stores are currently supported: an in-DRAM hash table and a MySQL database [6] paired with a Bloom filter. Since we implement a virtual layer upon the key-value store, it is easy to add a new key-value store.

3.2 Backup Pipeline

As shown in Figure 3, we divide the workflow of data deduplication into six phases: *Chunk*, *Hash*, *Dedup*, *Rewrite*, *Filter*, and *Append*. (1) The **Chunk** phase divides the backup stream into chunks. We have implemented Fixed-Sized Chunking and Content-Defined Chunking (CDC). (2) The **Hash** phase calculates a SHA-1 digest for each chunk as the fingerprint. (3) The **Dedup** phase aggregates chunks into segments, and identifies duplicate chunks via consulting the fingerprint index. A duplicate chunk is marked and obtains the container ID of its stored copy. The created segments are the prefetching units of logical locality, and the batch process units for physical locality. We have implemented the Base, Top-k, and *Mix* procedures (first Top-k then Base). (4) The **Rewrite** phase identifies fragmented duplicate chunks, and rewrites them to improve restore performance. It is a tradeoff between deduplication ratio and restore performance. We have implemented four rewriting algorithms, including CFL-SD [30], CBR [20], Capping [21], and HAR [17]. Each fragmented chunk is marked. (5) The **Filter** phase handles chunks according to their marks. Unique and fragmented chunks are added to the container buffer. Once the container buffer is full, it is pushed to the next phase. The recipe store and key-value store are updated. (6) The **Append** phase writes full containers to the container store.

We pipeline the phases via pthreads to leverage multi-core architecture. The dedup, rewrite, and filter phases are separated for modularity: we can implement a new rewriting algorithm without the need to modify the fingerprint index, and vice versa.

Segmenting and Sampling. The segmenting method is called in the dedup phase, and the sampling method is called for each segment either in the dedup phase for the similarity detection, or in the filter phase for the Base procedure. All segmenting and sampling methods mentioned in Section 2 have been implemented. Content-defined segmenting is implemented via checking the last n bits of a fingerprint. If all the bits are zero, the fingerprint (chunk) is considered to be the beginning of a new segment, thus generating an average segment size of 2^n chunks. To select the first fingerprint of a content-defined segment as a feature, the random sampling also checks the last $\log_2 R$ ($< n$) bits.

3.3 Restore Pipeline

The restore pipeline in DeFrame consists of three phases: *Reading Recipe*, *Reading Chunks*, and *Writing Chunks*. **(1) Reading Recipe**. The required backup recipe is opened for restore. The fingerprints are read and issued one by one to the next step. **(2) Reading Chunks**. Each fingerprint incurs a chunk read request. The container

Dataset name	Kernel	VMDK	RDB
Total size	104 GB	1.89 TB	1.12 TB
# of versions	258	127	212
Deduplication ratio	45.28	27.36	39.1
Avg. chunk size	5.29 KB	5.25 KB	4.5 KB
Self-reference	$< 1\%$	15-20%	0
Fragmentation	Severe	Moderate	Severe

Table 3: The characteristics of datasets.

is read from the container store to satisfy the request. A chunk cache is maintained to hold popular chunks in memory. We have implemented three kinds of restore algorithms, including the basic LRU cache, the optimal cache [17], and the rolling forward assembly area [21]. Given a chunk placement determined by the rewriting algorithm, a good restore algorithm boosts the restore procedure with a limited memory footprint. The required chunks are issued one by one to the next phase. **(3) Writing Chunks**. Using the received chunks, files are reconstructed in the local file system.

3.4 Garbage Collection

After users delete expired backups, chunks become invalid (not referenced by any backup) and must be reclaimed. There are a number of possible techniques for *garbage collection* (GC), such as reference counting [34] and mark-and-sweep [18]. Extending the DeFrame taxonomy to allow comparison of GC techniques is beyond the scope of this work; currently, DeFrame employs the History-Aware Rewriting (HAR) algorithm and Container-Marker Algorithm (CMA) proposed in [17]. HAR rewrites fragmented valid chunks to new containers during backups, and CMA reclaims old containers that are no longer referenced.

4 Evaluation

In this section, we evaluate the parameter space to find reasonable solutions that perform suitable tradeoffs.

4.1 Experimental Setup

We use three real-world datasets as shown in Table 3. Kernel is downloaded from the web [5]. It consists of 258 versions of unpacked Linux kernel source code. VMDK is from a virtual machine with Ubuntu 12.04. We compiled the source code, patched the system, and ran an HTTP server on the virtual machine. VMDK has many self-references; it also has less fragmentation from its fewer versions and random updates. RDB consists of Redis database [7] snapshots. The database has 5 million records, 5 GB in space and an on average 1% update ratio. We disable the default rdbcompression option.

All datasets are divided into variable-sized chunks via CDC. We use the content-defined segmenting with an average segment size of 1024 chunks by default. The container size is 4 MB, which is close to the average size of segments. The default fingerprint cache has 1024 slots to hold prefetching units, being either containers or segments. Hence, the cache can hold 1 million fingerprints, which is relatively large for our datasets.

4.2 Metrics and Our Goal

Our evaluations are in terms of quantitative metrics listed as follow. (1) *Deduplication ratio*: the original backup data size divided by the size of stored data. It indicates how efficiently data deduplication eliminates duplicates, being an important factor in the storage cost. (2) *Memory footprint*: the runtime DRAM consumption. A low memory footprint is always preferred due to DRAM's high unit price and energy consumption. (3) *Storage cost*: the cost for storing chunks and the fingerprint index, including memory footprint. We ignore the cost for storing recipes, since it is constant. (4) *Lookup requests per GB*: the number of required lookup requests to the key-value store to deduplicate 1 GB of data, most of which are random reads. (5) *Update requests per GB*: the number of required update requests to the key-value store to deduplicate 1 GB of data. A higher lookup/update overhead degrades the backup performance. Lookup requests to unique fingerprints are eliminated since most of them are expected to be answered by the in-memory Bloom filter. (6) *Restore speed*: 1 divided by mean containers read per MB of restored data [21]. It is used to evaluate restore performance, where a higher value is better. Since the container size is 4 MB, 4 units of restore speed translate to the maximum storage bandwidth.

It is practically impossible to find a solution that performs the best in all metrics. Our goal is to find some reasonable solutions with the following properties: (1) sustained, high backup performance as the top priority; (2) reasonable tradeoffs in the remaining metrics.

4.3 Exact Deduplication

Previous studies [39, 14] of EDPL fail to have an insight of the impacts of the fragmentation on the backup performance, since their datasets are short-term. Figure 4 shows the ever-increasing lookup overhead. We observe 6.5-12.0× and 5.1-114.4× increases in Kernel and RDB respectively under different fingerprint cache sizes. A larger cache cannot address the fragmentation problem; a 4096-slot cache performs as poor as the default 1024-slot cache. A 128-slot cache results in a 114.4× increase in RDB, which indicates an insufficient cache can result in unexpectedly poor performance. This causes complications in practice due to the difficulty in predicting how much memory is required to avoid unexpected performance degradations. Furthermore, even with a large cache, the lookup overhead still increases over time.

Before comparing EDLL to EDPL, we need to determine the best segmenting and sampling methods for EDLL. Figure 5(a) shows the lookup/update overheads

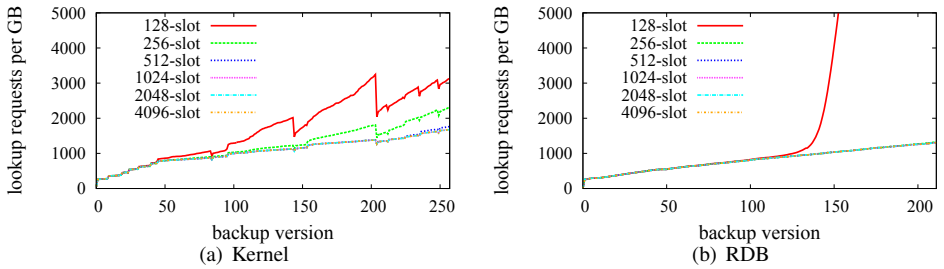

Figure 4: The ever-increasing lookup overhead of EDPL in Kernel and RDB under various fingerprint cache sizes.

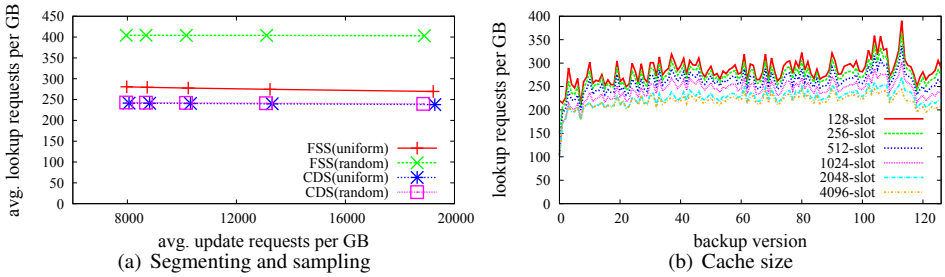

Figure 5: Impacts of varying segmenting, sampling, and cache size on EDLL in VMDK. **(a)** FSS is Fixed-Sized Segmenting and CDS is Content-Defined Segmenting. Points in a line are of different sampling ratios, which are 256, 128, 64, 32, and 16 from left to right. **(b)** EDLL is of CDS and a 256:1 random sampling ratio.

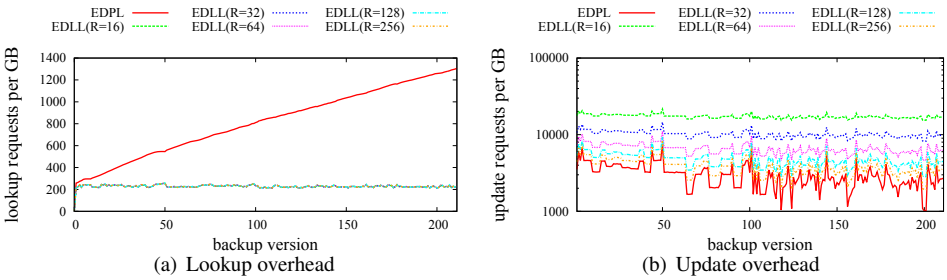

Figure 6: Comparisons between EDPL and EDLL in terms of lookup and update overheads. $R = 256$ indicates a sampling ratio of 256:1. Results come from RDB.

of EDLL under different segmenting and sampling methods in VMDK. Similar results are observed in Kernel and RDB. Increasing the sampling ratio shows an efficient tradeoff: a significantly lower update overhead at a negligible cost of a higher lookup overhead. The fixed-sized segmenting paired with the random sampling performs worst. This is because it cannot sample the first fingerprint in a segment, which is important for the Base procedure. The other three combinations are more efficient since they sample the first fingerprint (the random sampling performs well in the content-defined segmenting due to our optimization in Section 3.2). The content-defined segmenting is better than the fixed-sized segmenting due to its shift-resistance. Figure 5(b) shows the lookup overheads in VMDK under different cache sizes. We do not observe an ever-increasing trend of lookup overhead in EDLL. A 128-slot cache results in additional I/O (17% more than the default) due to the space-oriented similar segments in VMDK. Kernel and RDB (not shown in the figure) do not cause this problem because they have no self-reference.

Figure 6 compares EDPL and EDLL in terms of lookup and update overheads. EDLL uses the content-defined segmenting and random sampling. Results in Kernel and VMDK are not shown, because they have similar results to RDB. While EDPL suffers from the ever-increasing lookup overhead, EDLL has a much lower and sustained lookup overhead (3.6× lower than EDPL on average). With a 256:1 sampling ratio, EDLL has 1.29× higher update overhead since it updates sampled duplicate fingerprints with their new segment IDs. Note that lookup requests are completely random, and update requests can be optimized to sequential writes via a log-structured key-value store, which is a popular design [8, 23, 15]. Overall, if the highest deduplication ratio is required, EDLL is a better choice due to its sus-

tained high backup performance.

Finding (1): While the fragmentation results in an ever-increasing lookup overhead in EDPL, EDLL achieves sustained performance. The sampling optimization performs an efficient tradeoff in EDLL.

4.4 Near-exact Deduplication exploiting Physical Locality

NDPL is simple and easy to implement. Figure 7(a) shows how to choose an appropriate sampling method for NDPL. We only show the results from VMDK, which are similar to the results from Kernel and RDB. The uniform sampling achieves significantly higher deduplication ratio than the random sampling. The reason has been discussed in Section 2.1.2; for the random sampling, the missed duplicate fingerprints are definitely not sampled, making new containers have less features and hence smaller probability of being prefetched. The sampling ratio is a tradeoff between memory footprint and deduplication ratio: a higher sampling ratio indicates a lower memory footprint at a cost of a decreased deduplication ratio. Figure 7(b) shows that NDPL is surprisingly resistent to small cache sizes: a 64-slot cache results in only an 8% decrease of the deduplication ratio than the default in RDB. Also observed (not shown in the figure) are 24-93% additional I/O, which come from prefetching fingerprints. Compared to EDPL, NDPL has better backup performance because of its in-memory key-value store at a cost of decreasing deduplication ratio.

Finding (2): In NDPL, the uniform sampling is better than the random sampling. The fingerprint cache has minimal impacts on deduplication ratio.

4.5 Near-exact Deduplication exploiting Logical Locality

Figure 8(a) compares the Base procedure (see Figure 1) to the simplest similarity detection Top-1, which helps to choose appropriate sampling method. The content-defined segmenting is used due to its advantage shown in EDLL. In the Base procedure, the random sampling achieves comparable deduplication ratio using less memory than the uniform sampling. NDLL is expected to outperform NDPL in terms of deduplication ratio since NDLL does not suffer from fragmentation. However, we surprisingly observe that, while NDLL does better in Kernel and RDB as expected, NDPL is better in VMDK (shown in Figure 7(b) and 8(b)). The reason is that self-reference is common in VMDK. The fingerprint prefetching is misguided by space-oriented similar segments as discussed in Section 2.1.3. Moreover, the fingerprint cache contains many duplicate fingerprints that reduce the effective cache size, therefore a 4096-slot cache improves deduplication ratio by 7.5%. NDPL does not have this problem since its prefetching unit (i.e., container) is after-deduplication. A 64-slot cache results in 23% additional I/Os in VMDK (not shown in the figure), but has no side-effect in Kernel and RDB.

In the Top-1 procedure, only the most similar segment is read. The minimum sampling is slightly better than the random sampling. The Top-1 procedure is worse than the Base procedure. The reason is two-fold as discussed in Section 2.1.3: (1) a segment boundary change results in more time-oriented similar segments; (2) self-reference results in many space-oriented similar segments.

Finding (3): The Base procedure underperforms in NDLL if self-reference is common. Reading a single most similar segment is insufficient due to self-reference and segment boundary changes.

We further examine the remaining NDLL subspace: segment selection (s), segment prefetching (p), and mapping relationship (v). Figure 9 shows the impacts of varying the three parameters on deduplication ratio (lookup overheads are omitted due to space limits). On the X-axis, we have parameters in the format (s,p,v). The s indicates the segment selection method, being either Base or Top-k. The p indicates the number of prefetched segments plus the selected segment. We apply segment prefetching to all similar segments selected. The v indicates the maximum number of segments that a feature refers to. The random sampling is used, with a sampling ratio of 128. For convenience, we use NDLL(s,p,v) to represent a point in the space.

A larger v results in a higher lookup overhead when $k > 1$, since it provides more similar segment candidates. We observe that increasing v is not cost-effective in Kernel which lacks self-reference, since it increases lookup overhead without an improvement of deduplication ratio. However, in RDB which also lacks of self-reference, NDLL(Top-1,1,2) achieves better deduplication ratio than NDLL(Top-1,1,1) due to the rollbacks in RDB. A larger v is helpful to improve deduplication ratio in VMDK where self-reference is common. For example, NDLL(Top-1,1,2) achieves 1.31× higher deduplication ratio than NDLL(Top-1,1,1) without an increase of lookup overhead.

The segment prefetching is efficient for increasing deduplication ratio and decreasing lookup overhead. As the parameter p increases from 1 to 4 in the Base procedure, the deduplication ratios increase by 1.06×, 1.04×, and 1.39× in Kernel, RDB, and VMDK respectively, while the lookup overheads decrease by 3.81×, 3.99×, and 3.47×. The Base procedure is sufficient to achieve a high deduplication ratio in Kernel and RDB that lack self-reference. Given its simple logical frame, the Base procedure is a reasonable choice if self-reference is rare.

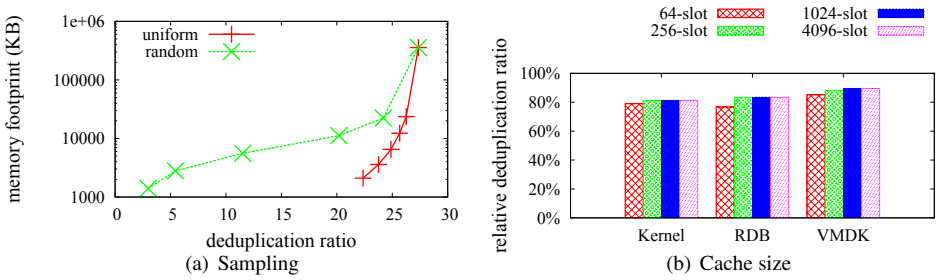

Figure 7: Impacts of varying sampling method and cache size on NDPL. **(a)** Points in each line are of different sampling ratios, which are 256, 128, 64, 32, 16, and 1 from left to right. **(b)** NDPL uses a 128:1 uniform sampling ratio. The Y-axis shows the relative deduplication ratio to exact deduplication.

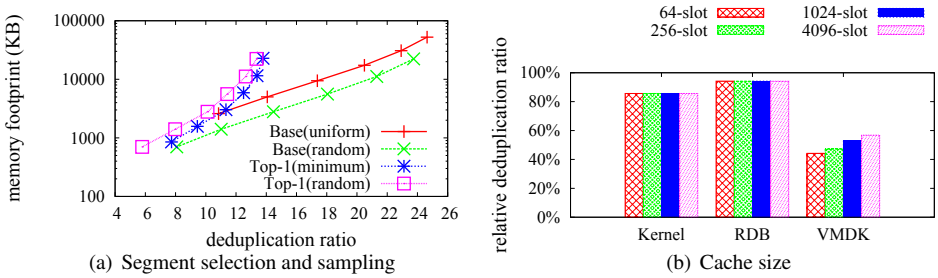

Figure 8: **(a)** Comparisons of Base and Top-1 in VMDK. Points in each line are of different sampling ratios, which are 512, 256, 128, 64, 32, and 16 from left to right. **(b)** NDLL is of the Base procedure and a 128:1 random sampling. The Y-axis shows the relative deduplication ratio to exact deduplication.

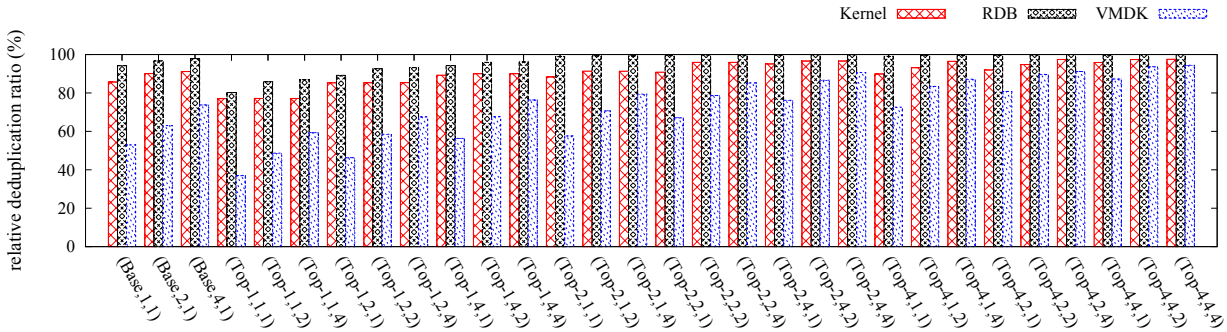

Figure 9: Impacts of the segment selection, segment prefetching, and mapping relationship on deduplication ratio. The deduplication ratios are relative to those of exact deduplication.

However, the Base procedure only achieves a 73.74% deduplication ratio of exact deduplication in VMDK where self-reference is common.

Finding (4): If self-reference is rare, the Base procedure is sufficient for a high deduplication ratio.

In more complicated environments like virtual machine storage, the Top-k procedure is desired. A higher k indicates a higher deduplication ratio at a cost of a higher lookup overhead. As k increases from NDLL(Top-1,1,1) to NDLL(Top-4,1,1), the deduplication ratios increase by 1.17×, 1.24×, and 1.97× in Kernel, RDB, and VMDK respectively, at a cost of 1.15×, 1.01×, and 1.56× more segment reads. Note that Top-4 outperforms Base in terms of deduplication ratio in all datasets. Varying k has fewer impacts in Kernel and RDB, since they have fewer space-oriented similar segments and hence fewer candidates. The segment prefetching is a great complement to the Top-k procedure, since it amortizes the additional lookup overhead caused by increasing k. NDLL(Top-4,4,1) reduces the lookup overheads of NDLL(Top-4,1,1) by 2.79×, 3.97×, and 2.07× in Kernel, RDB, and VMDK respectively. It also improves deduplication ratio by a factor of 1.2× in VMDK. NDLL(Top-4,4,1) achieves a 95.83%, 99.65%, and 87.20% deduplication ratio of exact deduplication, significantly higher than NDPL.

Finding (5): If self-reference is common, the similarity detection is required. The segmenting prefetching is a great complement to Top-k.

	Dataset	EDPL	NDPL-128	NDPL-256	NDPL-512	HAR+EDPL
Deduplication ratio	Kernel	45.35	36.86	33.52	30.66	31.26
	RDB	39.10	32.64	29.31	25.86	28.28
	VMDK	27.36	24.50	23.15	21.46	24.90
Restore speed	Kernel	0.50	0.92	1.04	1.19	2.60
	RDB	0.50	0.82	0.87	0.95	2.26
	VMDK	1.39	2.49	2.62	2.74	2.80

Table 4: Comparisons between near-exact deduplication and rewriting in terms of restore speed and deduplication ratio. NDPL-256 indicates NDPL of a 256:1 uniform sampling ratio. HAR uses EDPL as the fingerprint index. The restore cache contains 128 containers for Kernel, and 1024 containers for RDB and VMDK.

4.6 Rewriting Algorithm and its Interplay

Fragmentation decreases restore performance significantly in aged systems. The rewriting algorithm is proposed to trade deduplication ratio for restore performance. To motivate the rewriting algorithm, Table 4 compares near-exact deduplication to a rewriting algorithm, History-Aware Rewriting algorithm (HAR) [17]. We choose HAR due to its accuracy in identifying fragmentation. As the baseline, EDPL has best deduplication ratio and hence worst restore performance. NDPL shows its ability of improving restore performance, however not as well as HAR. Taking RDB as an example, NDPL of a 512:1 uniform sampling ratio trades 33.88% deduplication ratio for only 1.18× improvement in restore speed, while HAR trades 27.69% for 2.8× improvement.

We now answer the questions in Section 2.2: (1) How does the rewriting algorithm improve EDPL in terms of lookup overhead? (2) How does fingerprint index affect the rewriting algorithm? Figure 10(a) shows how HAR improves EDPL. We observe that HAR successfully stops the ever-increasing trend of lookup overhead in EDPL. Although EDPL still has a higher lookup overhead than EDLL, it is not a big deal because a predictable and sustained performance is the main concern. Moreover, HAR has no impact on EDLL, since EDLL does not exploit physical locality that HAR improves. The periodic spikes are because of major updates in Linux kernel, such as from 3.1 to 3.2. These result in many new chunks, which reduce logical locality. Figure 10(b) shows how fingerprint index affects HAR. EDPL outperforms EDLL in terms of deduplication ratio in all datasets. As explained in Section 2.2, EDLL could return an obsolete container ID if an old segment is read, and hence a recently rewritten chunk would be rewritten again. Overall, with an efficient rewriting algorithm, EDPL is a better choice than EDLL due to its higher deduplication ratio and sustained performance.

Finding (6): The rewriting algorithm helps EDPL to achieve sustained backup performance. With a rewriting algorithm, EDPL is better due to its higher deduplication ratio than other index schemes.

We further examine three restore algorithms: the LRU cache, the forward assembly area (ASM) [21], and the optimal cache (OPT) [17]. Figure 11 shows the efficiencies of these restore algorithms with and without HAR in Kernel and VMDK. Because the restore algorithm only matters under limited memory, the DRAM used is smaller than Table 4, 32-container-sized in Kernel and 256-container-sized in VMDK. If no rewriting algorithm is used, the restore performance of EDPL decreases over time due to the fragmentation. ASM has better performance than LRU and OPT, since it never holds useless chunks in memory. If HAR is used, EDPL has sustained high restore performance since the fragmentation has been reduced. OPT is best in this case due to its efficient cache replacement.

Finding (7): Without rewriting, the forward assembly area is recommended; but with an efficient rewriting algorithm, the optimal cache is better.

4.7 Storage Cost

As discussed in Section 2, indexing 1 TB unique data of 4 KB chunks in DRAM, called *baseline*, costs $140, 57.14% of which is for DRAM. The cost is even higher if considering the high energy consumption of DRAM. The baseline storage costs are $0.23, $3.11, and $7.55 in Kernel, RDB, and VMDK respectively.

To reduce the storage cost, we either use HDD instead of DRAM for exact deduplication or index a part of fingerprints in DRAM for near-exact deduplication. Table 5 shows the relative storage costs to the baseline in each dataset. EDPL and EDLL have the identical storage cost, since they have the same deduplication ratio and key-value store. We assume that the key-value store in EDPL and EDLL is a database paired with a Bloom filter, hence 1 byte DRAM per stored chunk for a low false positive ratio. EDPL and EDLL reduce the storage cost by a factor of around 1.75. The fraction of the DRAM cost is 2.27-2.50%.

Near-exact deduplication of a high sampling ratio further reduces the DRAM cost, at a cost of decreasing deduplication ratio. As discussed in Section 2.1.1, near-exact deduplication with a 128:1 sampling ratio and 4 KB chunk size needs to achieve 97% of deduplication ratio of exact deduplication to avoid a cost increase. To evaluate this tradeoff, we observe the storage costs of NDPL and NDLL under various sampling ratios. NDPL uses the

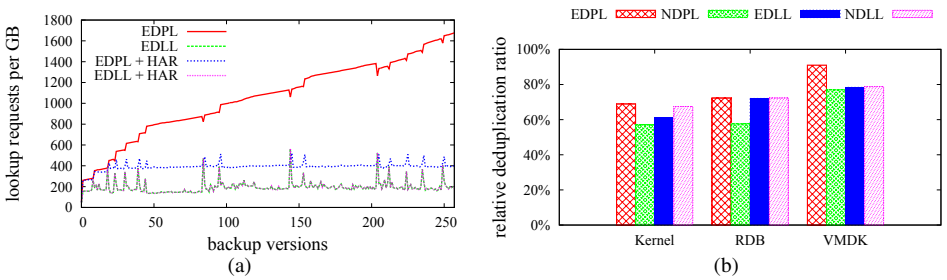

Figure 10: Interplays between fingerprint index and rewriting algorithm (i.e., HAR). **(a)** How does HAR improve EDPL in terms of lookup overhead in Kernel? **(b)** How does fingerprint index affect HAR? The Y-axis shows the relative deduplication ratio to that of exact deduplication without rewriting.

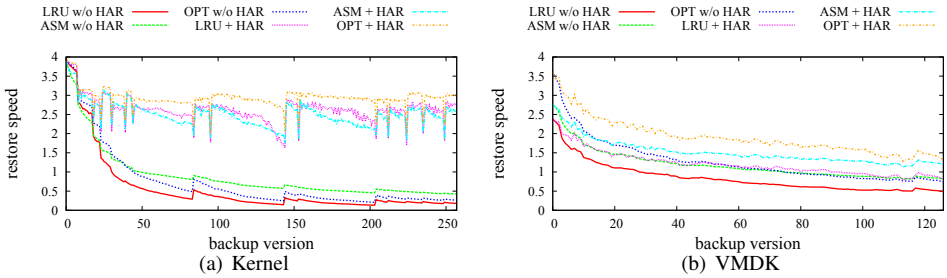

Figure 11: Interplays between the rewriting and restore algorithms. EDPL is used as the fingerprint index.

Dataset	*Fraction*	*EDPL/EDLL*	*NDPL-64*	*NDPL-128*	*NDPL-256*	*NDLL-64*	*NDLL-128*	*NDLL-256*
Kernel	DRAM	1.33%	0.83%	0.49%	0.31%	0.66%	0.34%	0.16%
	HDD	57.34%	65.01%	70.56%	77.58%	59.03%	59.83%	60.23%
	Total	58.67%	65.84%	71.04%	77.89%	59.69%	60.17%	60.39%
RDB	DRAM	1.40%	0.83%	0.48%	0.31%	0.70%	0.35%	0.17%
	HDD	55.15%	61.25%	66.08%	73.58%	55.27%	55.34%	55.65%
	Total	56.55%	62.07%	66.56%	73.89%	55.97%	55.69%	55.82%
VMDK	DRAM	1.41%	0.82%	0.45%	0.27%	0.71%	0.35%	0.18%
	HDD	54.86%	60.32%	63.16%	67.10%	59.79%	62.92%	71.24%
	Total	56.27%	61.14%	63.61%	67.36%	60.49%	63.27%	71.42%

Table 5: The storage costs relative to the baseline which indexes all fingerprints in DRAM. NDPL-128 is NDPL of a 128:1 uniform sampling ratio.

uniform sampling, and NDLL is of the parameter (Top-4,4,1). As shown in Table 5, NDPL increases the storage cost in all datasets; NDLL increases the storage cost in most cases, except in RDB.

Finding (8): Although near-exact deduplication reduces the DRAM cost, it cannot reduce the total storage cost.

5 Conclusions

In this paper, we discuss the parameter space of data deduplication in detail, and we present a general-purpose framework called DeFrame for evaluation. DeFrame can efficiently find reasonable solutions to explore tradeoffs among backup and restore performance, memory footprints, and storage costs. Our findings, from a large-scale evaluation using three real-world long-term workloads, provide a detailed guide to make efficient design decisions for deduplication systems.

It is impossible to have a solution that performs the best in all metrics, To achieve the lowest storage cost, Exact Deduplication exploiting Logical Locality (EDLL) is preferred due to its highest deduplication ratio and sustained high backup performance. To achieve the lowest memory footprint, Near-exact Deduplication is recommended: either exploiting Physical Locality (NDPL) for its simpleness, or exploiting Logical Locality (NDLL) for better deduplication ratio. To achieve a sustained high restore performance, Exact Deduplication exploiting Physical Locality (EDPL) combined with a rewriting algorithm would be a better choice.

Acknowledgments

We are grateful to our shepherd Fred Douglis and the anonymous reviewers for their insightful feedback. We appreciate Benjamin Young for his work to proofread the final version. The work was partly supported by

National Basic Research 973 Program of China under Grant No. 2011CB302301; NSFC No. 61025008, 61173043, 61232004, and 6140050892; 863 Project 2013AA013203; Fundamental Research Funds for the Central Universities, HUST, under Grant No. 2014QNRC019. The work conducted at VCU was partially sponsored by US National Science Foundation under Grants CNS-1320349 and CNS-1218960. The work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China.

References

[1] Amazon. http://www.amazon.com, 2014.

[2] Berkeley DB. http://www.oracle.com/us/products/database/berkeley-db/overview/index.htm, 2014.

[3] The digital universe of opportunities. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, 2014.

[4] Intel solid-state drive dc s3700 series. http://www.intel.com/content/www/us/en/solid-state-drives/solid-state-drives-dc-s3700-series.html, 2014.

[5] Linux kernel. http://www.kernel.org/, 2014.

[6] MySQL. http://www.mysql.com/, 2014.

[7] Redis. http://redis.io/, 2014.

[8] ANAND, A., MUTHUKRISHNAN, C., KAPPES, S., AKELLA, A., AND NATH, S. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. USENIX NSDI, 2010*.

[9] ARONOVICH, L., ASHER, R., BACHMAT, E., BITNER, H., HIRSCH, M., AND KLEIN, S. T. The design of a similarity based deduplication system. In *Proc. ACM SYSTOR, 2009*.

[10] BHAGWAT, D., ESHGHI, K., LONG, D. D., AND LILLIBRIDGE, M. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. IEEE MASCOTS, 2009*.

[11] BHATOTIA, P., RODRIGUES, R., AND VERMA, A. Shredder : Gpu-accelerated incremental storage and computation. In *Proc. USENIX FAST, 2012*.

[12] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM 13*, 7 (July 1970), 422–426.

[13] BRODER, A. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings* (Jun 1997), pp. 21–29.

[14] DEBNATH, B., SENGUPTA, S., AND LI, J. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proc. USENIX ATC, 2010*.

[15] DEBNATH, B., SENGUPTA, S., AND LI, J. SkimpyStash: Ram space skimpy key-value store on flash-based storage. In *Proc. ACM SIGMOD, 2011*.

[16] DONG, W., DOUGLIS, F., LI, K., PATTERSON, H., REDDY, S., AND SHILANE, P. Tradeoffs in scalable data routing for deduplication clusters. In *Proc. USENIX FAST, 2011*.

[17] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *Proc. USENIX ATC, 2014*.

[18] GUO, F., AND EFSTATHOPOULOS, P. Building a high-performance deduplication system. In *Proc. USENIX ATC, 2011*.

[19] HUANG, P., WU, G., HE, X., AND XIAO, W. An aggressive worn-out flash block management scheme to alleviate ssd performance degradation. In *Proc. ACM EuroSys, 2014*.

[20] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proc. ACM SYSTOR, 2012*.

[21] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. USENIX FAST, 2013*.

[22] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse indexing: large scale, inline deduplication using sampling and locality. In *Proc. USENIX FAST, 2009*.

[23] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. SILT: A memory-efficient, high-performance key-value store. In *Proc. ACM SOSP, 2011*.

[24] MANDAGERE, N., ZHOU, P., SMITH, M. A., AND UTTAMCHANDANI, S. Demystifying data deduplication. In *Proc. ACM/IFIP/USENIX Middleware, 2008*.

[25] MEISTER, D., KAISER, J., AND BRINKMANN, A. Block locality caching for data deduplication. In *Proc. ACM SYSTOR, 2013*.

[26] MEISTER, D., KAISER, J., BRINKMANN, A., CORTES, T., KUHN, M., AND KUNKEL, J. A study on data deduplication in hpc storage systems. In *Proc. IEEE SC, 2012*.

[27] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. In *Proc. USENIX FAST, 2011*.

[28] MIN, J., YOON, D., AND WON, Y. Efficient deduplication techniques for modern backup operation. *Computers, IEEE Transactions on 60*, 6 (June 2011), 824–840.

[29] NAM, Y., LU, G., PARK, N., XIAO, W., AND DU, D. H. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *Proc. IEEE HPCC, 2011*.

[30] NAM, Y. J., PARK, D., AND DU, D. H. Assuring demanded read performance of data deduplication storage with backup datasets. In *Proc. IEEE MASCOTS, 2012*.

[31] PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. A case for redundant arrays of inexpensive disks (RAID). In *Proc. ACM SIGMOD, 1988*.

[32] PAULO, J. A., AND PEREIRA, J. A survey and classification of storage deduplication systems. *ACM Comput. Surv. 47*, 1 (June 2014), 11:1–11:30.

[33] QUINLAN, S., AND DORWARD, S. Venti: a new approach to archival storage. In *Proc. USENIX FAST, 2002*.

[34] STRZELCZAK, P., ADAMCZYK, E., HERMAN-IZYCKA, U., SAKOWICZ, J., SLUSARCZYK, L., WRONA, J., AND DUBNICKI, C. Concurrent deletion in a distributed content-addressable storage system with global deduplication. In *Proc. USENIX FAST, 2013*.

[35] TARASOV, V., JAIN, D., KUENNING, G., MANDAL, S., PALANISAMI, K., SHILANE, P., TREHAN, S., AND ZADOK, E. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*.

[36] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of backup workloads in production systems. In *Proc. USENIX FAST, 2012*.

[37] WILDANI, A., MILLER, E. L., AND RODEH, O. Hands: A heuristically arranged non-backup in-line deduplication system. In *Proc. IEEE ICDE, 2013*.

[38] XIA, W., JIANG, H., FENG, D., AND HUA, Y. Silo: a similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proc. USENIX ATC, 2011*.

[39] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. USENIX FAST, 2008*.