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 tradeoff 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 solution can do best in all metrics, our goal is to find some reasonable solutions that have sustained performance and suitable tradeoff. Our findings from extensive experiments using real-world long-term workloads provide a detailed guide to make efficient design decisions according to required tradeoff.

1 Introduction

Efficient storage of a huge volume of digital data becomes a big challenge for industry and academia. ID-C predicts that there will be 44ZB 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 [29, 39], secondary backup systems [37], and high-performance data centers [28]. 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 its SHA-1 digest commonly referred to as a *fingerprint*. Two chunks with identical fingerprints are considered duplicates without requiring a byte-by-byte comparison. The probability of SHA-1 collisions is much smaller than that of hardware errors [35], being the reason 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 in-line data deduplication. The chunks with unique fingerprints that don't exist in the fingerprint index are aggregated into fixed-sized *containers* (e.g., 4MB), which are managed in a log-structure manner [41]. A *recipe* that consists of the fingerprint sequence of the backup is written for future data recovery.

There have been many publications [34] about data deduplication. 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, we present a taxonomy to disassemble existing work into individual design parameters, including **key-value**, **fingerprint prefetching and caching**, **segmenting**, **sampling**, **rewriting**, **restore**, etc. Different from previous surveys [26, 34], our taxonomy is fine-grained and has more 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 why existing solutions choose their points, which allows us to find potentially better solutions. For example, similarity detection in Sparse Index [23] and segment prefetching in SiLo [40] are highly complementary.

Although there are some open-source deduplication platforms, such as dmdedup [36], none of them are capable of evaluating the parameter space we discuss. Hence, the second part of our paper presents a general-purpose Deduplication Framework (**DeFrame**), for data deduplication evaluation. DeFrame implements the entire parameter space discussed, being modular and extensible for new algorithms; this enables apple-to-apple comparisons among existing and potential solutions. Our aim is to facilitate finding solutions that provide sustained high backup and restore performance, low memory footprint, and high storage efficiency.

The third part of our paper 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 required 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.

The contributions of this paper are (1) proposing *an N-dimensional parameter space* to characterize and classify data deduplication; (2) developing a general-purpose *framework*, which is capable of examining the large parameter space; and (3) an extensive *experimental evaluation* on the parameter space in terms of backup and restore performance, memory footprint, and storage efficiency using realistic long-term datasets.

The rest of the paper is organized as follows. Section 2 discusses the N-dimensional parameter space of data deduplication. Section 3 presents the framework. Section 4 evaluates the design tradeoff and highlights our findings. Section 5 gives related work and Section 6 concludes.

2 In-line Data Deduplication Space

Figure 1 depicts a typical deduplication system. Generally, we have three components on disks: (1) *Fingerprint index* maps fingerprints of stored chunks to their physical locations. It is used to identify duplicate chunks. (2) *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) *Container store* is a log-structure 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 hot 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 in our case (assuming a simplest segmenting approach). Each chunk is processed in the following steps: **(1)** 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, a fingerprint prefetching procedure is invoked. Jump to step (7). **(6)** Otherwise, the chunk is unique. We write the chunk to container store, insert the fingerprint to key-value store, and write the fingerprint to recipe store. Jump to step (1) to process next chunk. **(7)** The chunk is duplicate. We eliminate the chunk and write its fingerprint to recipe store. Jump to step (1) to process next chunk.

Table 1: The major parameters we discuss

Parameter list	Description
sampling	select representative fingerprints
segmenting	the unit of logical locality
segment selection	select segments to be prefetched
segment prefetching	exploit segment-level locality
key-value mapping	1:many mapping
rewriting	identify fragmentation
restore	choose restore algorithm

In following subsections, 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 [41]. The simplest fingerprint index is only a key-value store [35]. 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. Supposing 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 4KB on average, indexing 1TB unique data requires at least a 8GB-sized key-value store. To model the storage cost, we use the unit price from Amazon.com [1]: A Western Digital Blue 1TB 7200 RPM SATA Hard Drive takes $60, and a Kingston HyperX Blu 8GB 1600MHz DDR3 DRAM takes $80. The total storage cost is $140, 57.14% of which is for DRAM. Hence, putting all fingerprints in DRAM is not cost-efficient.

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 400MB/s under commercial CPUs [12], 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 HDD, i.e., 100 IOPS [16]. SSD supports much higher throughput, nearly

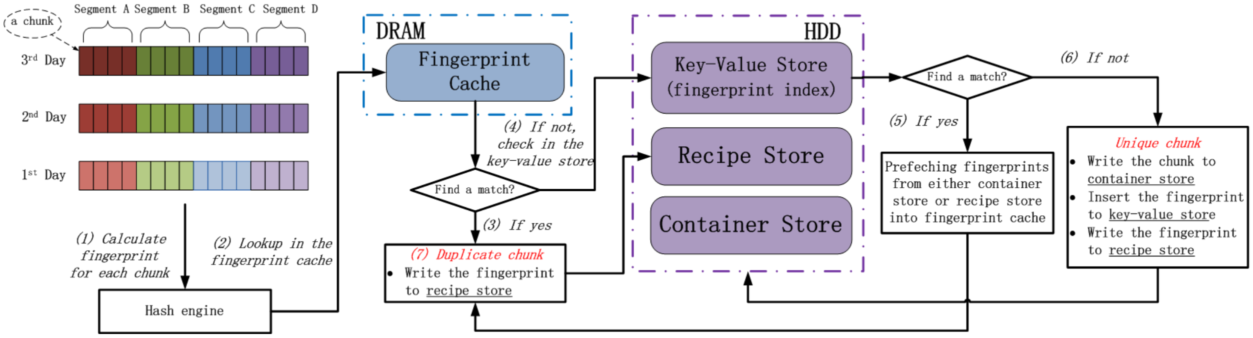

a chunk
Segment A
Segment B
Segment C
Segment D
3rd Day
2nd Day
1st Day
(1) Calculate fingerprint for each chunk
(2) Lookup in the fingerprint cache
Hash engine
DRAM
Fingerprint Cache
(4) If not, check in the key-value store
Find a match?
(3) If yes
(7) Duplicate chunk
Write the fingerprint to recipe store
HDD
Key-Value Store (fingerprint index)
Recipe Store
Container Store
Find a match?
(5) If yes
(6) If not
Prefeching fingerprints from either container store or recipe store into fingerprint cache
Unique chunk
Write the chunk to container store
Insert the fingerprint to key-value store
Write the fingerprint to recipe store

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

75,000 as venders report [4]. However, SSD is 10× more expensive than HDD and real-world experience suggests quite a lower IOPS.

Due to the incremental nature of backup workloads, the fingerprints of consecutive backups appear in similar sequences [41], 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**: all duplicate chunks are eliminated for highest deduplication ratio (the data size before deduplication divided by the data size after deduplication).

- **Near-exact deduplication**: 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**: the chunk (fingerprint) sequence of a backup stream, namely the chunk sequence before deduplication. It is preserved in recipes.

- **Physical locality**: the physical sequence of chunks (fingerprints), namely the chunk sequence after deduplication. It is preserved in containers.

Figure 2 shows the categories of existing fingerprint indexes, which are described in related work (Section 5). The fingerprint index space is hence divided into four subspaces: **EDPL**, **EDLL**, **NDPL**, and **NDLL**. In the following, we respectively discuss their parameter subspaces and how to choose reasonable parameter settings.

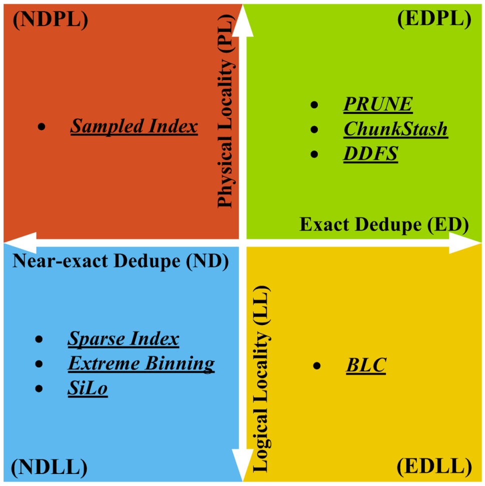


Figure 2: Categories of existing fingerprint indexes, including DDFS [41], Sparse Index [23], Extreme Binning [11], ChunkStash [16], Sampled Index [20], SiLo [40], PRUNE [30], and BLC [27].

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 to be not 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 can not be found in the key-value store. *The fingerprint prefetching/caching module is important to maintain high deduplication ratio.* Once an indexed duplicate fingerprint is found, many not-indexed fingerprints are prefetched to answer following lookup requests. The sampling method is important to the prefetching efficiency, and hence need 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 [13] as the key-value store, 1 byte DRAM per stored chunk is required to maintain a low false positive ratio for the Bloom filter. Supposing 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, the proportion of DRAM decreases.

The memory footprint of near-exact deduplication depends on the sampling ratio R. Supposing 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 say that near-exact deduplication saves money, since its stored data includes duplicate chunks. Supposing a 50% loss of deduplication ratio, 1TB data in near-exact deduplication only stores a half of 1TB data in exact deduplication. Hence, to decrease storage cost, near-exact deduplication has to achieve high deduplication ratio, no less than $(\frac{320}{S*R}+60)/(\frac{10}{S}+60)$ of 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. As a result, 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 [41]. 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 [31, 22]. 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 [20] 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. The uniform sampling doesn't have this problem, and hence achieves 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, which is different from 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 with 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

Table 2: Design choices for exploiting logical locality

	BLC [27]	**Extreme Binning [11]**	**Sparse Index [23]**	**SiLo [40]**
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

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 fingerprints pointing to stale segments may become unreachable that 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 [23, 11, 40] on NDLL use similarity detection (described later) instead of the Base procedure without explanation. 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 required 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 [11], which suffers from the greatly varied file size. The Fixed-Sized Segmenting (FSS) aggregates a fixed number (or size) of chunks into a segment [40]. 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 [23, 18]. 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 the Broder theory [14], 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*. Supposing the sampling ratio is R, the uniform sampling selects the first fingerprint every R fingerprints in a segment. The random sampling selects the fingerprints that $mod\ R = 0$ in a segment. 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. [10] propose to select the fingerprint adjacent to the minimum fingerprint. Based on our observations, the uniform sampling suffers from the problem of content shifting, while the random and minimum sampling are shift-resistant. 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 memory footprint. One feature per segment forces a single candidate.

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 to 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 [23]: (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 next similar segment, until k of segments are selected or we run out of all candidates. 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* [11], and thus a single I/O can fetch all similar segments. Hence, Top-*all* requires a dedicated bin store. It underperforms if we sample more than 1 feature per segment, since the bins grow big quickly.

At top left, 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 with 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 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 required 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*. 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. Supposing 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. Hence, E_2 is selected prior to A_2 when deduplicating A_3. A larger k increases the probability of reading A_2.

Segment prefetching. SiLo [40] exploits segment-level locality to prefetch segments. Supposing 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. In the case that more than 1 similar segments are read, segment prefetching can be applied to either all similar segments, as long as $k*p$ segments don't 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 above 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, 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) *Rollback occurs*. A rollback occurs when the corrupted primary data is restored to an earlier version rather than the latest version. 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 [19], which is an emerging dimension in the parameter space, was proposed to find fragmented duplicate chunks and rewrite them to new containers, allowing sustained high restore performance. 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 with 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 increase 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

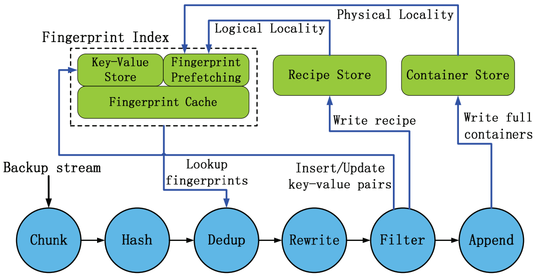


Figure 3: The architecture and backup pipeline

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) [22], and the optimal cache (OPT) [19]. 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 and OPT holds 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*[1] as a general-purpose chunk-level deduplication framework. In DeFrame, existing and potential solutions are considered as specific points in the N-dimensional parameter space. Figure 2 and Table 2 have shown how to configure existing solutions in DeFrame for apple-to-apple comparisons. DeFrame aims to facilitate finding reasonable solutions for required tradeoff. We implement DeFrame by C and pthreads on 64-bit Linux.

3.1 Architecture

As shown in Figure 3, DeFrame mainly consists of three submodules. The *container stor*e is a log-structure storage system. Variable-sized chunks are aggregated into fixed-sized containers. 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. A recipe contains the complete fingerprint sequence of a backup, which is used to restore that backup. The associated container IDs are stored beside 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* is for checking whether a chunk is a duplicate. It is backed by 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 [7] 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 [32], CBR [21], Capping [22], and HAR [19]. 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 multicore 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

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

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. Random sampling selects the fingerprints of the last $\log_2 R$ bits that are zero. Hence, the first fingerprint of a content-defined segment is a feature, which is important for the Base procedure.

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 is read from the container store to satisfy the request. A chunk cache is maintained to hold hot chunks in memory. We have implemented three kinds of restore algorithms, including the basic LRU cache, the optimal cache [19], and the rolling forward assembly area [22]. 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, a lot of chunks become invalid (not referenced by any backup) and need to be reclaimed for space saving. DeFrame currently supports a FIFO deletion scheme: older backup is deleted earlier. For example. if we set ”backup-retention-time” in the configuration file to 30, only the latest 30 backups would be retained at any moment. To reclaim invalid chunks, DeFrame employs History-Aware Rewriting algorithm (HAR) and Container-Marker Algorithm (CMA) proposed in [19]. HAR rewrites fragmented valid chunks to new containers during backups, and CMA reclaims old containers that are no longer referenced.

4 Searching in the Space

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

4.1 Experimental Setup

We use three real-world datasets as shown in Table 3. Kernel is downloaded from the web [6]. It consists of 258 versions of unpacked Linux kernel source code. VMDK is from a virtual machine installed Ubuntu 12.04LTS. We compiled the source code, patched the system, and ran an HTTP server on the virtual machine. Self-reference is common in VMDK. RDB consists of Redis database [8] snapshots. The database has 5 million records, 5GB in space and an on average 1% update ratio. We disable the default rdbcompression option. Since chunking is not our main concern, all datasets are divided into variable-sized chunks via CDC. VMDK has less fragmentation, since it has fewer versions and less random updates than Kernel and RDB.

Table 3: Characteristics of datasets

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

We use the content-defined segmenting with an average segment size of 1024 chunks by default. The container size is 4MB, 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 to affect 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 1GB 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 1GB 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 [22]. It is used to evaluate restore performance, where a higher value is better. Since the container size is 4MB, 4 units of restore speed translate to the maximum storage bandwidth.

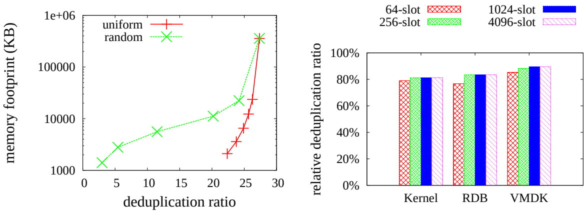
It is impossible to find a solution that does best in all metrics. Our goal is to find some reasonable solutions with the following properties: (1) sustained, high backup performance, which is always the highest priority; (2) reasonable tradeoff in remaining metrics.

4.3 Exact Deduplication

Previous studies [41, 16] 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× and 5.1–114.4× increases in Kernel and RDB respectively under different fingerprint cache sizes. A larger cache cannot solve 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 unexpectedly significant performance degradations. What's worse is that 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 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 higher lookup overhead. The fixed-sized segmenting paired with the random sampling performs worst, 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 don't 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) don't have 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 are similar to that in 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-structure key-value store, which is a popular design as described in Section 5. Overall, EDLL is a better choice if the highest deduplication ratio is required because of its sustained high backup performance.



(a) Sampling

(b) Cache size

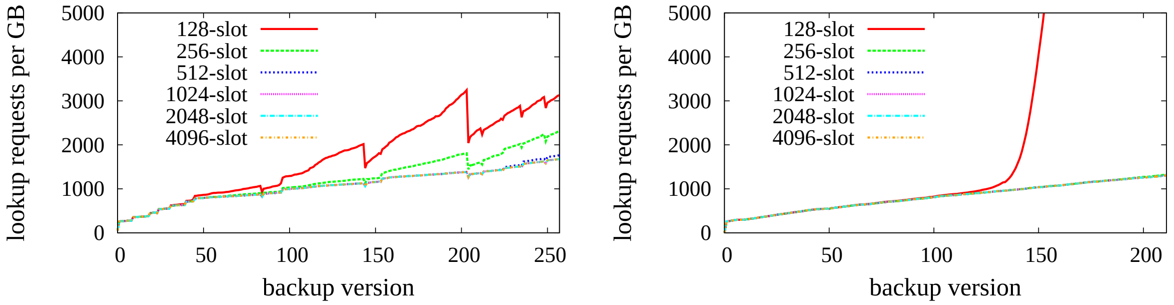
Figure 7: Impacts of varying sampling method and cache size on NDPL. **(a)** Points in each line are of different sampling ratios, which are 256, 128, 64, 32, 16, and 1 from left to right. **(b)** NDPL uses a 128:1 uniform sampling. The Y-axis shows the relative deduplication ratio to exact deduplication.

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 (but not sown 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.

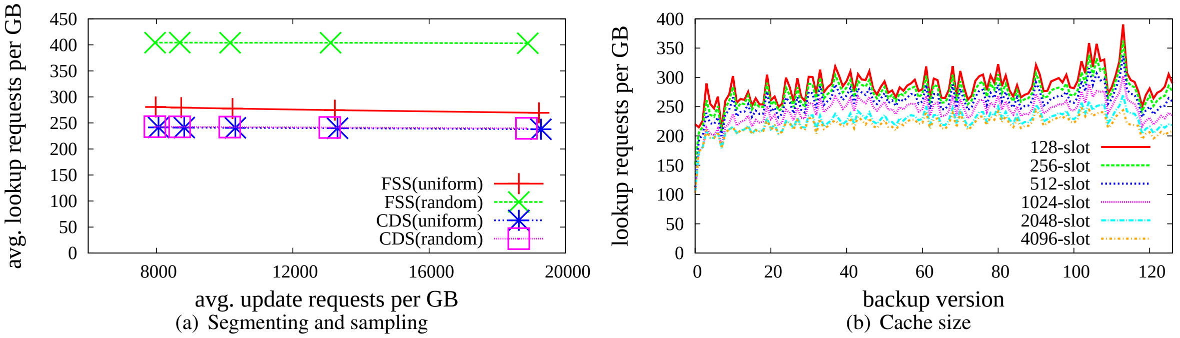
Finding (2): The uniform sampling does better in NDPL than the random sampling. The fingerprint cache affects backup performance more than deduplication ratio.



(a) Kernel

(b) RDB

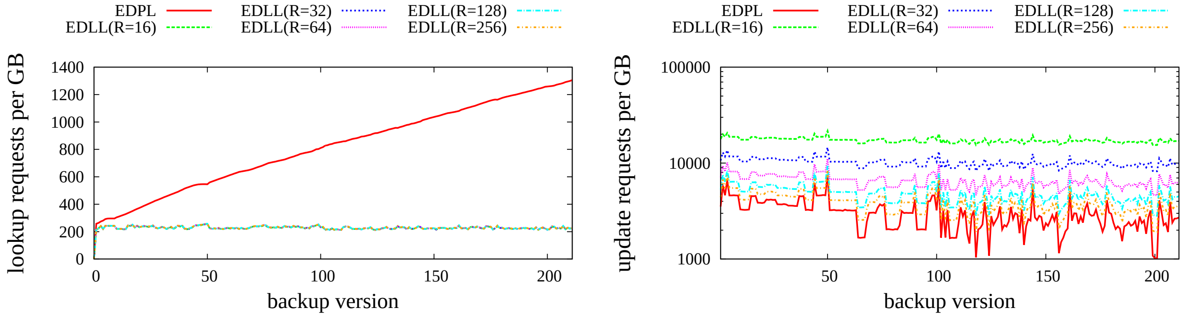
Figure 4: The ever-increasing lookup overhead of EDPL in Kernel and RDB under various fingerprint cache sizes.



(a) Segmenting and sampling

(b) Cache size

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.



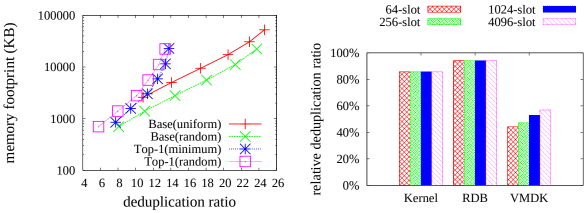
(a) Lookup overhead

(b) Update overhead

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.

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. Hence, the random sampling is better for the Base procedure in NDLL. NDLL is expected to outperform NDPL in terms of deduplication ratio since NDLL doesn't suffer from the 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. As a result, the fingerprint prefetching is misguided by space-oriented similar segments as discussed in Section 2.1.3. Additionally, the fingerprint cache contains many duplicate fingerprints that reduce the effective cache size. A 4096-slot cache improves deduplication ratio by 7.5% in VMDK. NDPL doesn't have this problem since its prefetching unit, namely container, is after-deduplication. A 64-slot cache results in 23% additional I/Os in VMDK (not shown), but no side-effect in Kernel and RDB.



(a) Segment selection and sampling (b) Cache size

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.

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 change.

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 and lookup overhead. 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 samping is used in both Base and Top-k, 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 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). This is because there exist some rollbacks in RDB. A larger v is helpful to improve deduplication ratio in VMDK where self-reference is common. For example, NDLL(1,1,2) achieves 1.31× higher deduplication ratio than NDLL(1,1,1) without an increase of lookup overhead.

The segment prefetching is very 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 in the datasets that lack self-reference, such as Kernel and RDB. Given its simple logical frame, the Base procedure is a reasonable choice if self-reference is rare. However, the Base procedure only achieves a 73.74% deduplication ratio of exact deduplication in VMDK where self-reference is common.

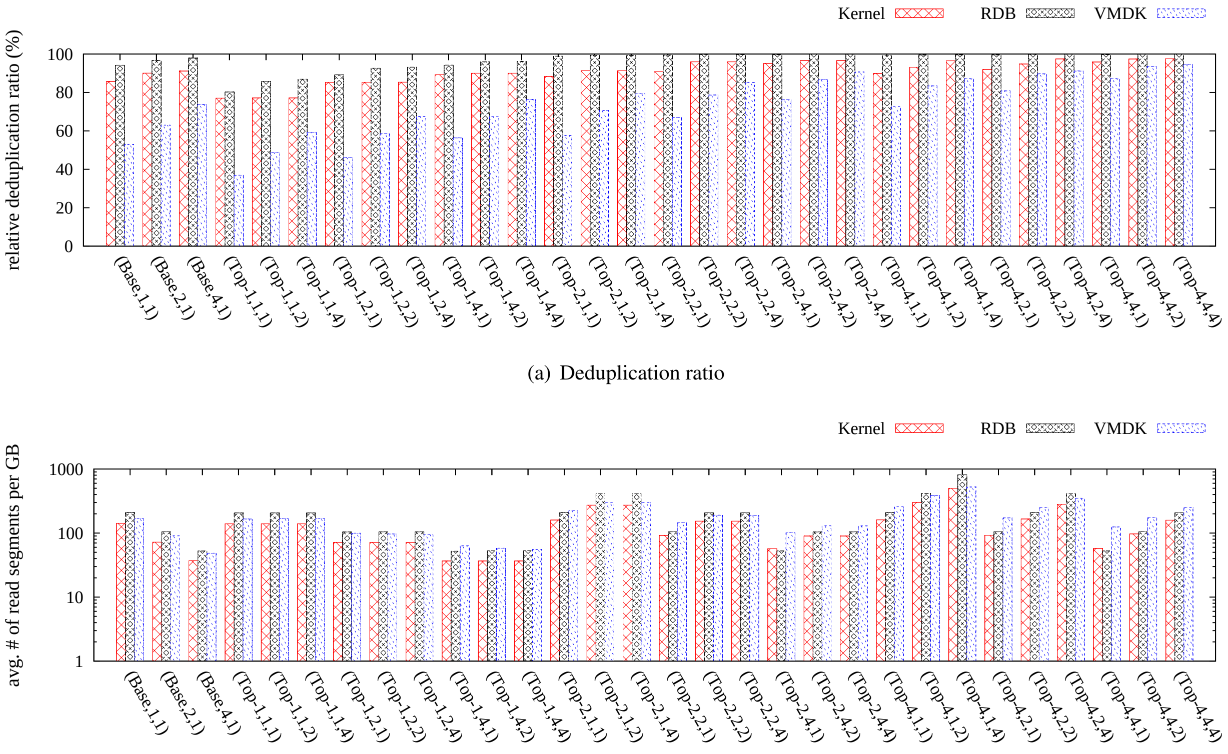
Finding (4): If self-reference is rare, the Base procedure is sufficient for high deduplication ratio.

In more complicated environment like virtual machine storage, the Top-k procedure is required. A higher k indicates higher deduplication ratio at a cost of 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 87.2% 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.

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) [19]. 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 trades 33.88% deduplication ratio for only 1.18× improvement in restore speed, while HAR trades 27.69% for 2.8× improvement.



(a) Deduplication ratio

(b) Lookup overhead

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

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. HAR uses EDPL as the fingerprint index. The restore cache is 128-, 1024-, and 1024-container-sized in Kernel, RDB, and VMDK.

	Dataset	*EDPL*	*NDPL-128*	*NDPL-256*	*NDPL-512*	*HAR+EDPL*
Deduplication ratio	Kernel	45.35	36.86	33.52	30.66	31.26
	RDB	39.1	32.64	29.31	25.86	28.28
	VMDK	27.36	24.5	23.15	21.46	24.9
Restore speed	Kernel	0.5	0.92	1.04	1.19	2.6
	RDB	0.5	0.82	0.87	0.95	2.26
	VMDK	1.39	2.49	2.62	2.74	2.8

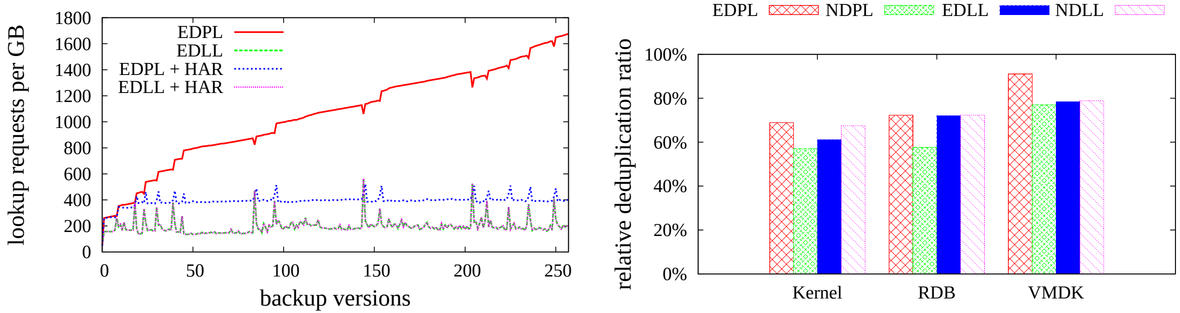


Figure 10: Interplays between fingerprint index and rewriting algorithm (i.e., HAR). **(a)** How does HAR improve EDPL in terms of lookup overhead in Kernel? **(b)** How does fingerprint index affect HAR? The Y-axis shows the relative deduplication ratio to that of exact deduplication without rewriting.

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 higher lookup overhead than EDLL, it is not a big deal because a predictable and sustained performance is the main concern. 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 have 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) [22], and the optimal cache (OPT) [19]. Figure 11 shows the efficiencies of these restore algorithms with and without HAR in Kernel and VMDK. The memory used is 32-container-sized in Kernel and 256-container-sized in VMDK, which are smaller than Table 4. It is because the restore algorithm only matters under limited memory. 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 1TB unique data of 4KB chunks requires a 8GB-sized key-value store. If we hold the key-value store in DRAM, namely the *baseline*, the storage cost is $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, and hence each stored chunk consumes 1 byte DRAM for a low false positive ratio. EDPL and EDLL significantly reduces the storage cost, by a factor of around 1.75×. The fraction of the DRAM cost is 2.27–2.5%.

Near-exact deduplication of a high sampling ratio further reduces the DRAM cost, at a cost of decreasing deduplication ratio. However, as discussed in Section 2.1.1, near-exact deduplication with a 128:1 sampling ratio and 4KB 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 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.

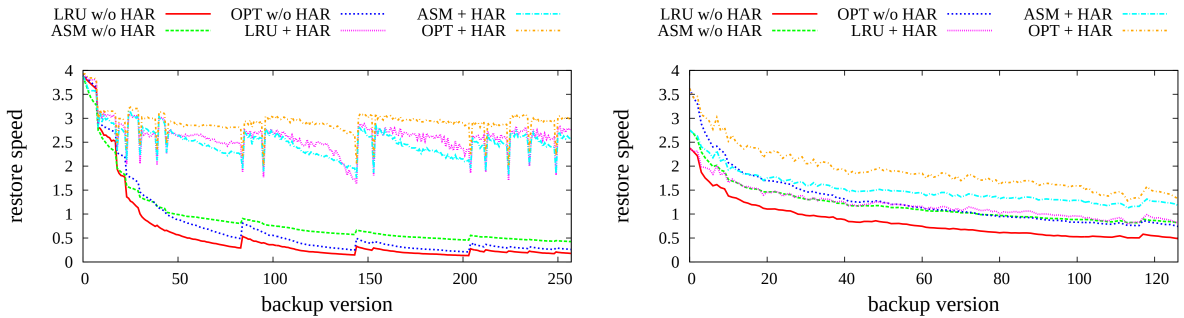
Finding (8): Although near-exact deduplication reduces the DRAM cost, it cannot reduce the total storage cost.

5 Related Work

Exact Deduplication. DDFS [41] uses a database paired with a Bloom filter as its key-value store. DDFS observes that containers preserve some locality, thus serving as good prefetch units of fingerprints. ChunkStash [16] follows the same prefetching scheme in DDFS, but uses an SSD-based key-value store. PRUNE [30] organizes its key-value store in a LRU-based log-structure to exploit physical locality. BLC [27] maintains a key-value store similar to DDFS for exact deduplication, but prefetches fingerprints in units of segments rather than containers.

Near-exact Deduplication. Sampled Index [20] samples the fingerprints in containers, and only the sampled fingerprints are indexed in DRAM. A fingerprint prefetching policy similar to DDFS guarantees the storage efficiency. Extreme Binning [11] selects a representative fingerprint in a file to find similar files. Data deduplication is only applied to similar files. Sparse Index [23] divides a backup stream into variable-sized segments. Representative fingerprints are randomly sampled within a segment. Data deduplication is only applied to most similar segments that share most representative fingerprints. SiLo [40] divides a backup stream into fixed-sized segments, and selects a representative fingerprint in a segment. SiLo observes and leverage the locality in adjacent segments to reduce I/Os.

Rewriting and Restore Algorithm. The rewriting algorithm aims to identify and rewrite fragmented duplicate chunks. CFL-SD [32], CBR [21], and Capping [22] use a rewriting buffer and identify fragmented chunks



(a) Kernel

(b) VMDK

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

Table 5: The storage costs relative to the baseline ($0.23, $3.11, $7.55 in Kernel, RDB, and VMDK respectively). NDPL-128 is NDPL of a 128:1 uniform sampling.

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	1.4%	0.83%	0.48%	0.31%	0.7%	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%	67.1%	59.79%	62.92%	71.24%
	Total	56.27%	61.14%	63.61%	67.36%	60.49%	63.27%	71.42%

Table 6: How to choose a reasonable solution according to required tradeoff.

Subspace	**Recommended parameter settings**	**Advantages**
EDLL	content-defined segmenting random sampling	lowest storage cost sustained backup performance
NDPL	uniform sampling	lowest memory footprint simplest logical frame
NDLL	content-defined segmenting similarity detection & segment prefetching	lowest memory footprint high deduplication ratio
EDPL	an efficient rewriting algorithm	sustained high restore performance good interplays with the rewriting algorithm

within the buffer. HAR [19] exploits historical information to identify fragmented chunks more accurately.

The forward assembly area [22] uses a large buffer as the assembly area, and reads containers one by one to assemble the buffer according to the recipe. The optimial restore cache [19] always evicts the cached container that will not be accessed for the longest time in the future via exploiting the recipe.

Key-value Store. Bigtable [15] and LevelDB [5, 38] use log-structured merge-tree (LSM-Tree) [33] to efficiently organize key-value pairs on disks and flash memory. BufferHash [9], SkimpyStash [17], SILT [24], and BloomStore [25] show how to design an efficient key-value store on flash memory. They all write key-value pairs in a log-structure manner, and are mainly concerned with the in-memory index structure that is a tradeoff between lookup performance and memory footprint.

6 Conclusions

In this paper, we disassemble data deduplication into an N-dimensional parameter space. Each point in the space represents a tradeoff among backup and restore performance, memory footprint, and storage cost. We then present a general-purpose framework called DeFrame to evaluate the parameter space. DeFrame aims to find reasonable solutions for required tradeoff. 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.

Although it is impossible to have a solution that performs best in all metrics, we summarize some reasonable solutions in Table 6. To achieve the lowest storage cost (DRAM and HDD), Exact Deduplication exploiting Logical Locality (EDLL) is preferred due to its highest deduplication ratio and sustained 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 sustained restore performance, a rewriting algorithm is required, and Exact Deduplication exploiting Physical Locality (EDPL) would be a better choice while combining with rewriting algorithm than other index schemes.

Acknowledgments

We are grateful to our shepherd Dr. Fred Douglis and the anonymous reviewers for their 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, and CQU, under Grant No. 106112013CDJZR180009; DPHEC under Grant No. 20130191120031; CBFR under Grant No. c-stc2013jcyjA40016. 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] Leveldb. code.google.com/p/leveldb, 2014.

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

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

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

[9] 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*.

[10] 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*.

[11] 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*.

[12] Bhatotia, P., Rodrigues, R., and Verma, A. Shredder : Gpu-accelerated incremental storage and computation. In *Proc. USENIX FAST, 2012*.

[13] Bloom, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13*, 7 (1970), 422–426.

[14] Broder, A. Z. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997. Proceedings* (1997), IEEE, pp. 21–29.

[15] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS) 26*, 2 (2008), 4.

[16] Debnath, B., Sengupta, S., and Li, J. ChunkStash: speeding up inline storage deduplication using flash memory. In *Proc. USENIX ATC, 2010*.

[17] Debnath, B., Sengupta, S., and Li, J. SkimpyStash: Ram space skimpy key-value store on flash-based storage. In *Proc. ACM SIGMOD, 2011*.

[18] Dong, W., Douglis, F., Li, K., Patterson, H., Reddy, S., and Shilane, P. Tadeoffs in scalable data routing for deduplication clusters. In *Proc. USENIX FAST, 2011*.

[19] 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*.

[20] Guo, F., and Efstathopoulos, P. Building a high-performance deduplication system. In *Proc. USENIX ATC, 2011*.

[21] Kaczmarczyk, M., Barczynski, M., Kilian, W., and Dubnicki, C. Reducing impact of data fragmentation caused by in-line deduplication. In *Proc. ACM SYSTOR, 2012*.

[22] Lillibridge, M., Eshghi, K., and Bhagwat, D. Improving restore speed for backup systems that use inline chunk-based deduplication. In *Proc. USENIX FAST, 2013*.

[23] Lillibridge, M., Eshghi, K., Bhagwat, D., Deolalikar, V., Trezise, G., and Camble, P. Sparse indexing: large s-cale, inline deduplication using sampling and locality. In *Proc. USENIX FAST, 2009*.

[24] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M. SILT: A memory-efficient, high-performance key-value store. In *Proc. ACM SOSP, 2011*.

[25] Lu, G., Nam, Y. J., and Du, D. H. BloomStore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash. In *Proc. IEEE MSST, 2012*.

[26] Mandagere, N., Zhou, P., Smith, M. A., and Uttamchandani, S. Demystifying data deduplication. In *Proc. ACM/IFIP/USENIX Middleware, 2008*.

[27] Meister, D., Kaiser, J., and Brinkmann, A. Block locality caching for data deduplication. In *Proc. ACM SYSTOR, 2013*.

[28] 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*.

[29] Meyer, D. T., and Bolosky, W. J. A study of practical deduplication. *ACM Transactions on Storage (TOS) 7*, 4 (2012), 14.

[30] Min, J., Yoon, D., and Won, Y. Efficient deduplication techniques for modern backup operation. *Computers, IEEE Transactions on 60*, 6 (2011), 824–840.

[31] 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*.

[32] 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*.

[33] O'NEIL, P. E., CHENG, E., GAWLICK, D., AND O'NEIL, E. J. The log-structured merge-tree (lsm-tree). *Acta Informatical 33* (1996), 351–385.

[34] PAULO, J., AND PEREIRA, J. A survey and classification of storage deduplication. *ACM Computing Surveys (CSUR) 47.1 (2014): 11.*

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

[36] 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*.

[37] 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*.

[38] WANG, P., SUN, G., JIANG, S., OUYANG, J., LIN, S., ZHANG, C., AND CONG, J. An efficient design and implementation of lsm-tree based key-value store on open-channel ssd. In *Proc. ACM EuroSys* (2014).

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

[40] 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*.

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