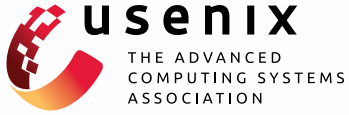


Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information

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, Fangting Huang, and Qing Liu,** *Huazhong University of Science and Technology*

https://www.usenix.org/conference/atc14/technical-sessions/presentation/fu_min

This paper is included in the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference.

June 18–20, 2014 • Philadelphia, PA

978-1-931971-10-2

Open access to the Proceedings of USENIX ATC '14: 2014 USENIX Annual Technical Conference is sponsored by USENIX.

Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information

Min Fu[†], Dan Feng[†✉], Yu Hua[†], Xubin He[‡], Zuoning Chen[*], Wen Xia[†], Fangting Huang[†], Qing Liu[†]
[†]*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*
[✉] *Corresponding author: dfeng@hust.edu.cn*

Abstract

In deduplication-based backup systems, the chunks of each backup are physically scattered after deduplication, which causes a challenging fragmentation problem. The fragmentation decreases restore performance, and results in invalid chunks becoming physically scattered in different containers after users delete backups. Existing solutions attempt to rewrite duplicate but fragmented chunks to improve the restore performance, and reclaim invalid chunks by identifying and merging valid but fragmented chunks into new containers. However, they cannot accurately identify fragmented chunks due to their limited rewrite buffer. Moreover, the identification of valid chunks is cumbersome and the merging operation is the most time-consuming phase in garbage collection.

Our key observation that fragmented chunks remain fragmented in subsequent backups motivates us to propose a History-Aware Rewriting algorithm (HAR). HAR exploits historical information of backup systems to more accurately identify and rewrite fragmented chunks. Since the valid chunks are aggregated in compact containers by HAR, the merging operation is no longer required. To reduce the metadata overhead of the garbage collection, we further propose a Container-Marker Algorithm (CMA) to identify valid containers instead of valid chunks. Our extensive experimental results from real-world datasets show HAR significantly improves the restore performance by 2.6X–17X at a cost of only rewriting 0.45–1.99% data. CMA reduces the metadata overhead for the garbage collection by about 90X.

1 Introduction

Deduplication has become a key component in modern backup systems due to its demonstrated ability of improving storage efficiency [26, 6]. A deduplication-based backup system divides a backup stream into variable-sized chunks [13], and identifies each chunk by its SHA-1 digest [19], i.e., *fingerprint*. A *fingerprint index* is used to map fingerprints of stored chunks to their physical addresses. In general, small and variable-sized chunks (e.g., 8KB on average [26]) are managed at a larger unit called *container* [26, 7, 9] that is a fixed-sized (e.g., 4MB [26]) structure. The containers are the basic unit of read and write operations. During a backup, the chunks that need to be written are aggregated into containers to preserve the locality of the backup stream. During a restore, a *recipe* (i.e., the fingerprint sequence of a backup) is read, and the containers serve as the prefetching unit. A restore cache holds the prefeteched containers and evicts an entire container via an LRU algorithm [9].

Since duplicate chunks are eliminated between multiple backups, the chunks of a backup unfortunately become physically scattered in different containers, which is known as fragmentation [18, 14]. First, the fragmentation severely decreases restore performance [15, 9]. The infrequent restore is important and the main concern from users [17]. Moreover, data replication, which is important for disaster recovery [20], requires reconstructions of original backup streams from deduplication systems [16], and thus suffers from a performance problem similar to the restore operation.

Second, the fragmentation results in invalid chunks (not referenced by any backups) becoming physically scattered in different containers when users delete expired backups. Existing solutions (i.e., reference management [7, 24, 4]) identify valid chunks and the containers holding only a few valid chunks. A merging operation is required to copy the valid chunks in the identified containers to new containers [10, 11], and then the identified containers are reclaimed. The merging is the most time-consuming phase in garbage collection [4].

A comprehensive category is helpful to understand the fragmentation. We observe that the fragmentation comes in two categories of containers: sparse containers and out-of-order containers. During a restore, a majority of chunks in a sparse container are never accessed, and the chunks in an out-of-order container are accessed inter-

mittently. Both of them hurt the restore performance. Increasing the restore cache size alleviates the negative impacts of out-of-order containers, but it is ineffective for sparse containers because they directly amplify read operations (read many never accessed chunks). Additionally, the merging operation is required to reclaim sparse containers in the garbage collection after users delete backups.

Reducing sparse containers is important to address the fragmentation problem. Existing solutions [15, 8, 9] propose to rewrite duplicate but fragmented chunks during the backup via *rewriting algorithms*, which is a trade-off between deduplication ratio (the size of the non-deduplicated data divided by that of the deduplicated data) and restore performance. These approaches buffer a small part of the backup stream, and identify the fragmented chunks within the buffer. They fail to identify sparse containers because an out-of-order container seems sparse in the limited-sized buffer. Hence, most of their rewritten chunks belong to out-of-order containers, which limit their gains in restore performance and garbage collection efficiency.

Our key observation is that two consecutive backups are very similar, and thus historical information collected during the backup is very useful to improve the next backup. For example, sparse containers for the current backup possibly remain sparse for the next backup. This observation motivates our work to propose a History-Aware Rewriting algorithm (HAR). During a backup, HAR rewrites the duplicate chunks in the sparse containers identified by the last backup, and records the emerging sparse containers to rewrite them in the next backup. HAR outperforms existing rewriting algorithms in terms of both restore performance and deduplication ratio. We also develop two optimization approaches for HAR to reduce the negative impacts of out-of-order containers on the restore performance, including an efficient restore caching scheme and a hybrid rewriting algorithm.

During the garbage collection, we need to identify valid chunks for identifying and merging sparse containers, which is cumbersome and error-prone due to the existence of large amounts of chunks. Since HAR efficiently reduces sparse containers, the identification of valid chunks is no longer necessary. We further propose a new reference management approach called Container-Marker Algorithm (CMA) that identifies valid containers (holding some valid chunks) instead of valid chunks. Comparing with existing reference management approaches, CMA significantly reduces the metadata overhead.

The paper makes the following contributions.

- We observe that the fragmentation is classified into two categories: out-of-order and sparse containers. The former reduces restore performance, which can be addressed by increasing the restore cache size. The latter reduces both restore performance and garbage collection efficiency, and we require a rewriting algorithm that is capable of accurately identifying sparse containers.

- In order to accurately identify and reduce sparse containers, we observe that sparse containers remain sparse in next backup, and hence propose HAR. HAR significantly improves restore performance with a slight decrease of deduplication ratio.

- In order to reduce the metadata overhead of the garbage collection, we propose CMA that identifies valid containers instead of valid chunks in the garbage collection.

The rest of the paper is organized as follow. Section 2 describes related work. Section 3 illustrates how the fragmentation arises. Section 4 discusses the fragmentation category and our observations. Section 5 presents our design and optimizations. Section 6 evaluates our approaches. Finally we conclude our work in Section 7.

2 Related Work

A deduplication system employs a large key-value subsystem, namely *fingerprint index*, to identify duplicates. The fingerprint index is too large to be completely stored in memory. However, a disk-based index that offers large-sized storage capacity suffers from severe performance bottleneck of accessing the fingerprints [19]. In order to address the performance problem of the fingerprint index, Zhu et al. [26] propose to leverage the locality of backup streams to accelerate fingerprint lookups. Extreme Binning [3], Sparse Index [10], and SiLo [25] mainly eliminate duplicate chunks among similar super-chunks (consists of many chunks). ChunkStash [5] stores the index in SSDs instead of disks.

The fragmentation problem in deduplication systems has received many attentions. iDedup [21] eliminates sequential and duplicate chunks in the context of primary storage systems. Nam et al. propose a quantitative metric to measure the fragmentation level of deduplication systems [14], and a selective deduplication scheme [15] for backup workloads. The Context-Based Rewriting algorithm (CBR) [8] and the capping algorithm (CAP) [9] are recently proposed to address the fragmentation problem.

CBR uses a fixed-sized buffer, called *stream context*, to maintain the following chunks of the pending duplicate chunk that is being determined whether fragmented. CBR defines the *rewrite utility* of a pending chunk as the size of the chunks that are in the *disk context* (physically adjacent chunks) but not in the stream context, divided by the size of the disk context. If the rewrite utility of

Table 1: Existing reference management approaches.

Offline	Perfect Hash Vector [4]
Inline	Reference Counter [24], Grouped Mark-and-Sweep [7]

the pending chunk is higher than the predefined *minimal rewrite utility*, the chunk is fragmented. CBR uses a *rewrite limit* to avoid too many rewrites.

CAP divides the backup stream into fixed-sized segments, and conjectures the fragmentation within each segment. CAP limits the maximum number (say T) of containers a segment can refer to. Suppose a new segment refers to N containers and $N > T$, the chunks in the $N - T$ containers that hold the least chunks in the segment are rewritten.

Both of CBR and CAP buffer a small part of the ongoing backup stream during a backup, and identify fragmented chunks within the buffer (generally 10-20MB). They fail to accurately identify fragmented chunks, since physically adjacent chunks of a duplicate chunk can be accessed beyond the buffer. Increasing the buffer size alleviates this problem but is not scalable. Our approach is based on a new observation that fragmented chunks remain fragmented in the next backup, hence accurately identifying fragmented chunks.

Reference management for the garbage collection is complicated in deduplication systems, because each chunk can be referenced by multiple backups. Existing reference management approaches are summarized in Table 1. The offline approaches traverse all fingerprints (including the fingerprint index and recipes) when the system is idle. For example, Botelho et al. [4] build a perfect hash vector as a compact representation of all chunks. Since recipes need to occupy significantly large storage space [12], the traversing operation is time-consuming. The inline approaches maintain additional metadata during backup to facilitate the garbage collection. Maintaining a reference counter for each chunk [24] is expensive and error-prone [7]. Grouped Mark-and-Sweep (GMS) [7] uses a bitmap to mark which chunks in a container are used by a backup.

3 The Fragmentation Problem

Deduplication improves storage efficiency but causes fragmentation [18, 14], which exacerbates restore performance and garbage collection efficiency. Figure 1 illustrates an example of two consecutive backups to show how the fragmentation arises. There are 13 chunks in the first backup. Each chunk is identified by a character, and duplicate chunks share an identical character. Two duplicate chunks, say A and D, are identified by deduplicating the stream, which is called *self-reference*. A and D are called *self-referred chunks*. All unique chunks are stored in the first 4 containers, and a blank is appended to the $4th$ half-full container to make it be aligned. With

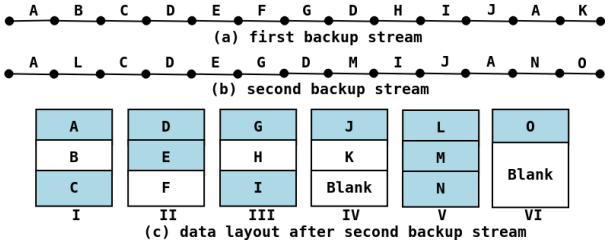


Figure 1: An example of two consecutive backups. The shaded areas in each container represent the chunks required by the second backup.

a 3-container-sized LRU cache, restoring the first backup needs to read 5 containers. The self-referred chunk A requires extra reading container I.

We observe that the second backup contains 13 chunks, 9 of which are duplicates in the first backup. The four new chunks are stored in two new containers. With a 3-container-sized LRU cache, restoring the second backup needs to read 9 containers.

Although both of the backups consist of 13 chunks, restoring the second backup needs to read 4 more containers than restoring the first backup. Hence, the restore performance of the second backup is much worse than that of the first backup. Recent work [15, 8, 9] also reported the severe decrease of restore performance in deduplication systems. We observe a $21X$ decrease in our Linux dataset (detailed in Section 6.2).

If we delete the first backup, several chunks including chunk K in container IV become invalid. Because chunk J is still referenced by the second backup, we can't reclaim container IV. Existing work [10, 11] uses the offline container merging operation. The merging reads the containers that have only a few valid chunks and copies them to new containers. Therefore, it suffers from a performance problem similar to the restore operation, thus becoming the most time-consuming phase in the garbage collection [4].

4 Fragmentation Classification and Our Observations

We observe that the fragmentation comes in two categories: sparse containers and out-of-order containers. In this section, we describe these two types of containers and their impacts, and then present our key observations that motivate our work.

4.1 Sparse Container

As shown in Figure 1, only one chunk in container IV is referenced by the second backup. Prefetching container IV for chunk J is inefficient when restoring the second backup. After deleting the first backup, we require a merging operation to reclaim the invalid chunks in container IV. This kind of containers exacerbates system performance on both restore and garbage collection. We define a container's *utilization* for a backup as the fraction

of its chunks referenced by the backup. If the utilization of a container is smaller than a predefined *utilization threshold*, such as 50%, the container is considered as a *sparse container* for the backup. We use the *average utilization* of all the containers related with a backup to measure the overall sparse level of the backup.

Sparse containers directly amplify read operations. Prefetching a container of 50% utilization at most achieves 50% of the maximum storage bandwidth, because 50% of the chunks in the container are never accessed. Hence, the average utilization determines the *maximum restore performance* with an unlimited restore cache. The chunks that have never been accessed in sparse containers require the slots in the restore cache, thus decreasing the available cache size. Therefore, reducing sparse containers can improve the restore performance.

After backup deletions, invalid chunks in a sparse container fail to be reclaimed until all other chunks in the container become invalid. Symantec [22] reports the probability that all chunks in a container become invalid is low. We also observe that garbage collection reclaims little space without additional mechanisms, such as offline merging sparse containers. Since the merging operation suffers from a performance problem similar to the restore operation, we require a more efficient solution to migrate valid chunks in sparse containers.

4.2 Out-of-order Container

If a container is accessed many times intermittently during a restore, we consider it as an *out-of-order container* for the restore. As shown in Figure 1, container V will be accessed 3 times intermittently while restoring the second backup. With a 3-container-sized LRU restore cache, restoring each chunk in container V incurs a cache miss that decreases restore performance.

The problem caused by out-of-order containers is complicated by self-references. The self-referred chunk *D* improves the restore performance, since the two accesses to *D* occur close in time. However, the self-referred chunk *A* decreases the restore performance.

The impacts of out-of-order containers on restore performance are related to the restore cache. For example, with a 4-container-sized LRU cache, restoring the three chunks in container V incurs only one cache miss. For each restore, there is a minimum cache size, called *cache threshold*, which is required to achieve the maximum restore performance (defined by the average utilization). Out-of-order containers reduce restore performance if the cache size is smaller than the cache threshold. They have no negative impact on garbage collection.

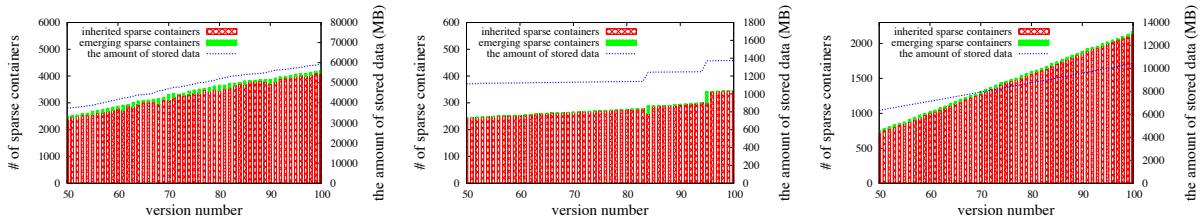
A sufficiently large cache can address the problem caused by out-of-order containers. However, since the memory is expensive, a restore cache of larger than the cache threshold can be unaffordable in practice. Hence, it is necessary to either decrease the cache threshold or assure the demanded restore performance if the cache is relatively small. If restoring a chunk in a container incurs an extra cache miss, it indicates that other chunks in the container are far from the chunk in the backup stream. Moving the chunk to a new container offers an opportunity to improve restore performance. Another more cost-effective solution to out-of-order containers is to develop a more intelligent caching scheme than LRU.

4.3 Our Observations

Because out-of-order containers can be alleviated by the restore cache, how to reduce sparse containers becomes the key problem. Existing rewriting algorithms cannot accurately identify sparse containers due to the limited buffer. Accurately identifying sparse containers requires the complete knowledge of the on-going backup. However, the complete knowledge of a backup cannot be known until the backup has concluded, making the identification of sparse containers a challenge.

Due to the incremental nature of backup, two consecutive backups are very similar, which is the major assumption behind DDFS [26]. Hence, they share similar characteristics, including the fragmentation. We analyze three datasets, including virtual machines, Linux kernels, and a synthetic dataset (detailed in Section 6.2), to explore and exploit potential characteristics of sparse containers (the utilization threshold is 50%). After each backup, we record the accumulative amount of the stored data, as well as the total and emerging sparse containers for the backup. An *emerging sparse container* is not sparse in the last backup but becomes sparse in the current backup. An *inherited sparse container* is already sparse in the last backup and remains sparse in the current backup. The total sparse containers are the sum of emerging and inherited sparse containers.

The characteristics of sparse containers are shown in Figure 2. First, the number of total sparse containers continuously grows. It indicates sparse containers become more common over time. Second, the number of total sparse containers increases smoothly most of time. A few exceptions in the Kernel datasets are major revision updates, which have more new data and increase the amount of stored data sharply. It indicates that a large update results in more emerging sparse containers. However, due to the similarity between consecutive backups, the number of emerging sparse containers of each backup is relatively small most of time. Third, the number of inherited sparse containers of each backup is equivalent to or slightly less than the number of total sparse containers of the previous backup. A few sparse containers of the previous backup become not sparse to the current backup since their utilizations drop to 0. It seldom occurs that the utilization of an inherited sparse container increases



(a) VMDK (b) Linux (c) Synthetic

Figure 2: Characteristics of sparse containers in three datasets. 50 backups are shown for clarity.

in the current backup, unless a rare rollback occurs. The observation indicates that sparse containers of the backup remain sparse in the next backup.

The above observations motivate our work to exploit the historical information to identify sparse containers. After completing a backup, we can determine which containers are sparse within the backup. Because these sparse containers remain sparse for the next backup, we record these sparse containers and allow chunks in them to be rewritten in the next backup. In such a scheme, the emerging sparse containers of a backup become the inherited sparse containers of the next backup. Due to the second observation, each backup needs to rewrite the chunks in a small number of inherited sparse containers, which would not degrade the backup performance. Moreover a small number of emerging sparse containers left to the next backup would not degrade the restore performance of the current backup. From the third observation, the scheme identifies sparse containers accurately. This scheme is called History-Aware Rewriting algorithm (HAR).

5 Design and Implementation

5.1 Architecture Overview

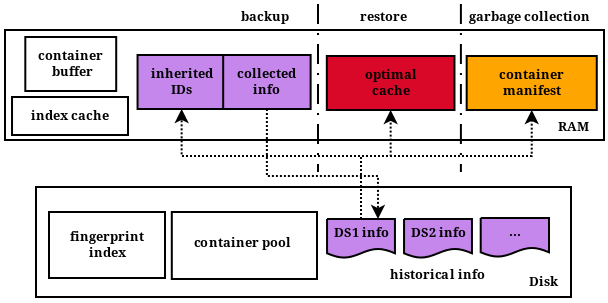


Figure 3: The HAR architecture.

Figure 3 illustrates the overall architecture of our HAR system. On disks, we have a container pool to provide container storage service. Any kinds of fingerprint indexes can be used. Typically we keep the complete fingerprint index on disks, as well as the hot part in memory. An in-memory container buffer is allocated for chunks to be written.

The system assigns each dataset a globally unique ID, such as $DS1$ in Figure 3. The collected historical information of each dataset is stored on disks with the dataset's ID, such as the $DS1\ info$ file. The collected historical information consists of three parts: IDs of inherited sparse containers for HAR, the container-access sequence for the Belady's optimal replacement cache, and the container manifest for Container-Marker Algorithm.

5.2 History-Aware Rewriting Algorithm

At the beginning of a backup, HAR loads IDs of all inherited sparse containers to construct the in-memory $S_{inherited}$ structure, and rewrites all duplicate chunks in the inherited sparse containers. In practice, HAR maintains two in-memory structures, S_{sparse} and S_{dense} (included in *collected info* in Figure 3), to collect IDs of emerging sparse containers. The S_{sparse} traces the containers whose utilizations are smaller than the utilization threshold. The S_{dense} records the containers whose utilizations exceed the utilization threshold. The two structures consist of *utilization records*, and each record contains a container ID and the current utilization of the container. After the backup is completed, HAR replaces the IDs of the old inherited sparse containers with the IDs of emerging sparse containers in S_{sparse}. Hence, the S_{sparse} becomes the $S_{inherited}$ of the next backup. The complete workflow of HAR is described in Algorithm 1.

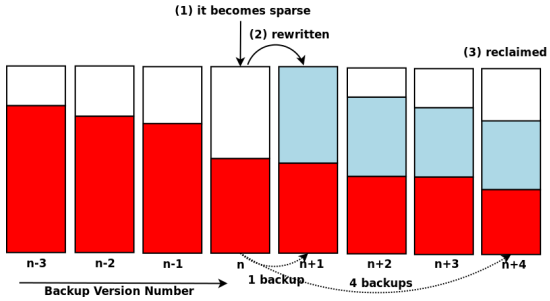


Figure 4: The lifespan of a rewritten sparse container.

Figure 4 illustrates the lifespan of a rewritten sparse container. The rectangle is a container, and the blank area is the chunks not referenced by the backup. We assume 4 backups are retained. (1) The container becomes sparse in backup n. (2) The container is rewritten in backup $n+1$. The chunks referenced by backup $n+1$ are rewritten to a new container that holds unique chunks and other

Algorithm 1 History-Aware Rewriting Algorithm

Input: IDs of inherited sparse containers, $S_{inherited}$;
Output: IDs of emerging sparse containers, S_{sparse};

```
Initialize two sets, S_sparse and S_dense.
while the backup is not completed do
    Receive a chunk and look up its fingerprint in the fingerprint index.
    if the chunk is duplicate then
        if the chunk's container ID exists in S_inherited then
            Rewrite the chunk, and obtain a new container ID.
        else
            Eliminate the chunk.
        end if
    else
        Write the chunk, and obtain a new container ID.
    end if
    if the chunk's container ID doesn't exist in S_dense then
        Update the associated utilization record (add it if doesn't exist) in S_sparse with the chunk size.
        if the utilization exceeds the utilization threshold then
            Move the utilization record to S_dense.
        end if
    end if
end while
return S_sparse
```

rewritten chunks (blue area). However the old container cannot be reclaimed after backup $n+1$, because backup $n-2$, $n-1$, and n still refer to the old container. (3) After backup $n+4$ is finished, all backups referring to the old container have been deleted, and thus the old container can be reclaimed. Each sparse container decreases the restore performance of the backup recognizing it, and will be reclaimed when the backup is deleted.

Due to the limited number of inherited sparse containers, the memory consumed by the $S_{inherited}$ is negligible. S_{sparse} and S_{dense} consume more memory because they need to monitor all containers related with the backup. If the default container size is 4MB and the average utilization is 50% which can be easily achieved by HAR, the two sets of a 1TB stream consume 8MB memory (each record contains a 4-byte ID, a 4-byte current utilization, and an 8-byte pointer). This analysis shows that the memory footprint of HAR is low in most scenarios.

There is a tradeoff in HAR. A higher utilization threshold results in more containers being considered sparse, and thus backups are of better average utilization and restore performance but worse deduplication ratio. If the utilization threshold is set to 50%, HAR promises an average utilization of no less than 50%, and the maximum restore performance is no less than 50% of the maximum storage bandwidth.

5.2.1 The Impacts of HAR on Garbage Collection

We define C_i as the set of containers related with backup i, $|C_i|$ as the size of C_i, n_i as the number of inherited sparse containers, r_i as the size of rewritten chunks, and d_i as the size of new chunks. T backups are retained at any moment. The container size is S. The storage cost can be measured by the number of valid containers. A container is valid if it has chunks referenced by non-deleted backups. After backup k is finished, the number of valid containers is N_k.

$$N_k = |\bigcup_{i=k-T+1}^{k} C_i| = |C_{k-T+1}| + \sum_{i=k-T+2}^{k} (\frac{r_i + d_i}{S})$$

For those deleted backups (before backup $k-T+1$), we have

$$|C_{i+1}| = |C_i| - n_{i+1} + \frac{r_{i+1} + d_{i+1}}{S}, 0 \le i < k-T+1$$

$$\Rightarrow N_k = |C_0| - \sum_{i=1}^{k-T+1} (n_i - \frac{r_i + d_i}{S}) + \sum_{i=k-T+2}^{k} (\frac{r_i + d_i}{S})$$

C_0 is the initial backup. Since the $|C_0|$, d_i, and S are constants, we concentrate on the part δ related with HAR,

$$\delta = -\sum_{i=1}^{k-T+1} (n_i - \frac{r_i}{S}) + \sum_{i=k-T+2}^{k} (\frac{r_i}{S}) \qquad (1)$$

The value of δ demonstrates the additional storage cost of HAR. If HAR is disabled (the utilization threshold is 0), δ is 0. A negative value of δ indicates that HAR decreases the storage cost. If k is small (the system is in the warn-up stage), the latter part is dominant thus HAR introduces additional storage cost than no rewriting. If k is large (the system is aged), the former part is dominant thus HAR decreases the storage cost.

A higher utilization threshold indicates that both n_i and r_i are larger. If k is small, a lower utilization threshold is helpful to decrease the storage cost since the latter part is dominant. Otherwise, the best utilization threshold is related with the backup retention time and characteristics of datasets. For example, if backups never expire, a higher utilization threshold always results in higher storage cost. Only retaining 1 backup would yield the opposite effect. However we find a value of 50% works well according to our experimental results in Section 6.7.

5.3 Optimal Restore Cache

To reduce the negative impacts of out-of-order containers on restore performance, we implement Belady's optimal replacement cache [2]. Implementing the optimal cache (OPT) needs to know the future access pattern. We can

collect such information during the backup, since the sequence of reading chunks during the restore is just the same as the sequence of writing them during a backup.

After a chunk is processed through either elimination or over-writing its container ID, its container ID is known. We add an *access record* into the collected info in Figure 3. Each access record can only hold a container ID. Sequential accesses to the identical container can be merged into a record. This part of historical information can be updated to disks periodically, and thus would not consume much memory.

At the beginning of a restore, we load the container-access sequence into memory. If the cache is full, we evict the cached container that will not be accessed for the longest time in the future. Belady has proven the optimality [2].

The complete sequence of access records can consume considerable memory when out-of-order containers are dominant. Assuming each container is accessed 50 times intermittently and the average utilization is 50%, the complete sequence of access records of a 1TB stream consumes over 100MB of memory. Instead of checking the complete sequence of access records, we can use a slide window to check a fixed-sized part of the future sequence, as a near-optimal scheme. The memory footprint of this near-optimal scheme is hence bounded. Because the recent backups are most likely restored [8], we only maintain the sequences of a few recent backups for storage savings, and restore earlier backups via an LRU replacement caching scheme.

5.4 A Hybrid Scheme

As discussed in Section 4.2, rewriting chunks in out-of-order containers offers opportunities to reduce their negative impacts. Since most of the chunks rewritten by existing rewriting algorithms belong to out-of-order containers, we propose a hybrid scheme that takes advantages of both HAR and existing rewriting algorithms (e.g., CBR [8] and CAP [9]) as optional optimizations. The hybrid scheme is straightforward. Each duplicate chunk not rewritten by HAR is further examined by CBR or CAP. If CBR or CAP considers the chunk fragmented, the chunk is rewritten.

To avoid a significant decrease of deduplication ratio, we configure CBR or CAP to rewrite less data than the exclusive uses of themselves. For example, CBR uses a *rewrite limit* to control the rewrite ratio (the size of the rewritten chunks divided by that of the total chunks). The default rewrite limit in CBR is 5%, and thus CBR attempts to rewrite top-5% fragmented chunks. Generally a higher rewrite limit indicates CBR rewrites more data for higher restore performance. We set rewrite limit to 0.5% in the hybrid of HAR and CBR. The hybrid of HAR and CAP is similar. Based on our observations, only rewriting a small number of additional chunks further improves restore performance when the restore cache is small. However, the hybrid scheme always rewrites more data than HAR. Hence, we propose disabling the hybrid scheme if a large restore cache is affordable (Since restore is rare and critical, a large cache is reasonable).

5.5 Container-Marker Algorithm

Existing garbage collection schemes rely on merging sparse containers to reclaim invalid chunks in the containers. Before merging, they have to identify invalid chunks to determine utilizations of containers, i.e., reference management. Existing reference management approaches [24, 7, 4] are inevitably cumbersome due to the existence of large amounts of chunks.

HAR naturally accelerates expirations of sparse containers and thus the merging is no longer necessary. Hence, we need not to calculate the exact utilization of each container. We design the Container-Marker Algorithm (CMA) to efficiently determine which containers are invalid. CMA is fault-tolerant and recoverable.

CMA maintains a *container manifest* for each dataset. The container manifest records IDs of all containers related to the dataset. Each ID is paired with a backup time, and the backup time indicates the dataset's most recent backup that refers to the container. Each backup time can be represented by one byte, and let the backup time of the earliest non-deleted backup be 0. One byte suffices differentiating 256 backups, and more bytes can be allocated for longer backup retention time. Each container can be used by many different datasets. For each container, CMA maintains a dataset list that records IDs of the datasets referring to the container. A possible approach is to store the lists in the blank areas of containers, which on average is half of the chunk size. After a backup is completed, the backup time of the containers whose IDs are in the S_{sparse} and S_{dense} are updated to the largest time in the old manifest plus one. CMA adds the dataset's ID to the lists of the containers that are in the new manifest but not in the old one. If the lists (or manifests) are corrupted, we can recover them by traversing manifests of all datasets (or all related recipes).

If we need to delete the oldest t backups of a dataset, CMA loads the container manifest into memory. The container IDs with a backup time smaller than t are removed from the manifest, and the backup time of the remaining IDs decreases by t. CMA removes the dataset's ID from the lists of the removed containers. If a container's list is empty, the container can be reclaimed. We further examine the fingerprints in reclaimed containers. If a fingerprint is mapped to a reclaimed container in the fingerprint index, its entry is removed.

Because HAR effectively maintains high utilizations of containers, the container manifest is small. We as-

Table 2: Characteristics of datasets.

dataset name	VMDK	Linux	Synthetic
total size	1.44TB	104GB	4.5TB
# of versions	102	258	400
deduplication ratio	25.44	45.24	37.26
avg. chunk size	10.33KB	5.29KB	12.44KB
sparse	medium	severe	severe
out-of-order	severe	medium	medium

sume that each backup is 1TB and 90% identical to adjacent backups. Recent 20 backups are retained. With a 50% average utilization, the backups at most refer to 1.5 million containers. Hence the manifest and lists consume at most 13.5MB storage space (each container has a 4-byte container ID paired with a 1-byte backup time in the manifest, and a 4-byte dataset ID in its list).

6 Performance Evaluation

6.1 Experimental Configurations

We implemented an experimental platform to evaluate our design, including HAR, OPT, and CMA. We also implement CBR [8] (The original CBR is designed for HydraStor [6], and we implement the idea in the container storage), CAP [9], and their hybrid schemes (HAR+CBR and HAR+CAP) for comparisons. Since the design of fingerprint index is out of scope for the paper, we simply accommodate the complete fingerprint index in memory. The *baseline* has no rewriting, and the default caching scheme is OPT. The container size is 4MB. The default utilization threshold in HAR is 50%. We retain 20 backups thus backup $n-20$ is deleted after backup n is finished. We don't apply the offline container merging as in previous work [15, 9], because it requires a long idle time.

We use Speed Factor [9] as the metric of the restore performance. The speed factor is defined as 1 divided by mean containers read per MB of restored data. Higher speed factor indicates better restore performance. Given the container size is 4MB, 4 units of speed factor correspond to the maximum storage bandwidth.

6.2 Datasets

Two real-world datasets, including VMDK and Linux, and a synthetic dataset, i.e., Synthetic, are used for evaluation. Their characteristics are listed in Table 2. Each dataset is divided into variable-sized chunks.

VMDK is from a virtual machine installed Ubuntu 12.04LTS, which is a common use-case in real-world [7]. We compile source code, patch the system, and run an HTTP server on the virtual machine. We backup the virtual machine regularly. It consists of 102 full backups. Each full backup is 14.48GB on average, and 90–98% identical to its adjacent backups. Each backup contains about 15% self-referred chunks, and thus out-of-order containers are dominant.

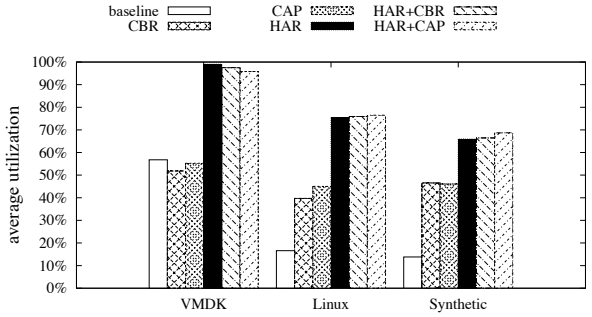


Figure 5: The average utilization of last 20 backups achieved by each rewriting algorithm.

Linux, downloaded from the web[1], is a commonly used public dataset [23]. It consists of 258 consecutive versions of unpacked Linux kernel sources. Each version is 412.78MB on average. Two consecutive versions are generally 99% identical except when there are large upgrades. In Linux, there are only a few self-references and sparse containers are dominant.

Synthetic is generated according to existing approaches [23, 9]. We simulate common operations of file systems, such as create/delete/modify files. We finally obtain a 4.5TB dataset with 400 versions. There is no self-reference in Synthetic.

6.3 Average Utilization

The average utilization of a backup exhibits its maximum restore performance. Figure 5 shows the average utilizations of rewriting algorithms. We observe that HAR significantly improves average utilizations, and obtains highest average utilizations in all datasets. The average utilizations of HAR are 99%, 75.42%, and 65.92% in VMDK, Linux, and Synthetic respectively, which indicate the *maximum speed factors* ($=$ *average utilization* $*$ 4) are 3.96, 3.02, and 2.64. CBR and CAP achieve lower average utilizations than the baseline in VMDK, because they rewrite many copies of self-referred chunks. They improve the average utilizations in Linux and Synthetic, although less than HAR by 30–50%. The hybrid schemes achieve average utilizations similar to HAR's.

6.4 Deduplication Ratio

Deduplication ratio explains the amount of written chunks, and the storage cost if no backup is deleted. Since we delete backups regularly to triggers garbage collection, the actual storage cost is shown in Section 6.6.

Figure 6 shows deduplication ratios of rewriting algorithms. The deduplication ratios of HAR are 22.78, 27.78, and 21.38 in VMDK, Linux, and Synthetic respectively. HAR rewrites 11.66%, 62.83%, and 74.31% more data than the baseline. However, the corresponding rewrite ratios remain at a low level, respectively 0.45%, 1.38%, and 1.99%. It indicates the size of rewritten

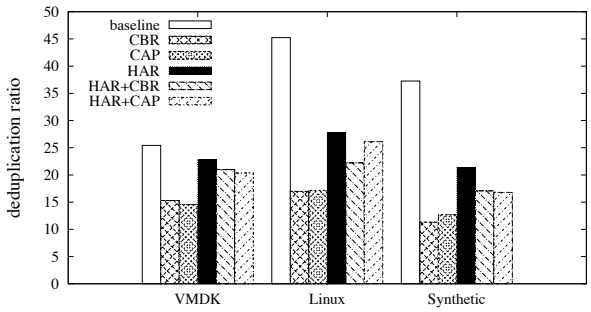


Figure 6: The comparisons between HAR and other rewriting algorithms in terms of deduplication ratio.

data is small relative to the size of backups. Due to such low rewrite ratios, the fingerprint lookup, content-defined chunking, and SHA-1 computation remain the performance bottleneck. Hence, HAR has trivial impacts on the backup performance.

We observe that HAR achieves considerably higher deduplication ratios than CBR and CAP. Since the rewrite ratios of CBR and CAP are 2 times larger than that of HAR, it is reasonable to expect that HAR outperforms CBR and CAP in terms of backup performance. The hybrid schemes, HAR+CBR and HAR+CAP, achieve better deduplication ratio than CBR and CAP respectively, but decrease deduplication ratios compared with HAR, such as by 10% in VMDK.

6.5 Restore Performance

Figure 7 shows the restore performance achieved by each rewriting algorithm with a given cache size. We tune the cache size according to the datasets, and show the impacts of varying cache size later in Figure 8. The default caching scheme is OPT. We observe severe declines of the restore performance in the baseline. For instance, restoring the latest backup is 21X slower than restoring the first backup in Linux. OPT alone increases restore performance by 1.51X, 1.47X, and 1.88X respectively in last 20 backups, however the performance remains at a low level.

We further examine the average speed factor in last 20 backups of each rewriting algorithm. In VMDK, CBR and CAP further improve restore performance by 1.46X and 1.53X respectively based on OPT. HAR outperforms them and increases restore performance by a factor of 1.72. The hybrid schemes are efficient, because HAR+CBR and HAR+CAP increase restore performance by 1.2X and 1.3X based on HAR. Given that their deduplication ratios are slightly smaller than HAR, CBR and CAP are good complements to HAR in the datasets where out-of-order containers are dominant. The restore performance of the initial backups exceeds the maximum storage bandwidth (4 units of speed factor), because self-referred chunks in the scope of the cache improve restore performance.

In Linux, CBR and CAP further improve restore performance by 5.4X and 6.12X. HAR is more efficient and further increases restore performance by a factor of 10.25. Because out-of-order containers are less dominant, the hybrid schemes can't achieve significantly better performance than HAR. Thus the hybrid schemes can be disabled in the datasets where the problem of out-of-order containers is less severe. There are some occasional smaller values in the curve of HAR, because a large upgrade in Linux kernel produces a large amount of sparse containers.

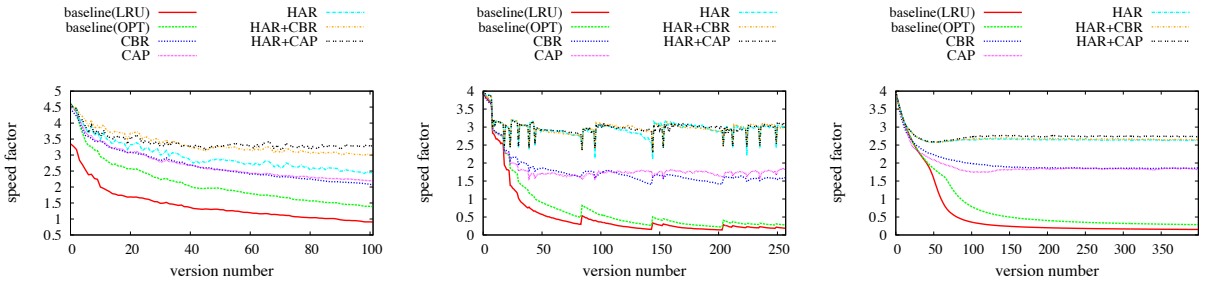
The results in Synthetic are similar with those in Linux. CBR, CAP, and HAR further increase restore performance by 6.41X, 6.35X, and 9.08X respectively. The hybrid schemes can't outperform HAR remarkably.

Figure 8 compares restore performance among rewriting algorithms under various cache sizes. In VMDK, because out-of-order containers are dominant, HAR requires a large cache (e.g., 2048-container-size) to achieve the maximum restore performance. We observe that if the cache size continuously increases, the restore performance of the baseline is approximate to that of CBR and CAP. The reason is that the baseline, CBR, and CAP achieve similar average utilizations as shown in Figure 5. CBR and CAP are great complements to HAR. When the cache is small, the restore performance of HAR+CBR (HAR+CAP) is approximate to that of CBR (CAP); when the cache is large, the restore performance of the hybrid schemes is approximate to that of HAR. Compared with HAR, the hybrid schemes successfully decrease the cache threshold by nearly 2X, and improve the restore performance when the cache is small.

In Linux, HAR achieves better restore performance than CBR and CAP, even with a small cache (e.g., 8-container-size). Compared with HAR, the hybrid schemes decrease the cache threshold by a factor of 2, and improve the restore performance when the cache is small. However, because the cache threshold of HAR is small, a restore cache of reasonable size can address the problem caused by out-of-order containers without decreasing deduplication ratio.

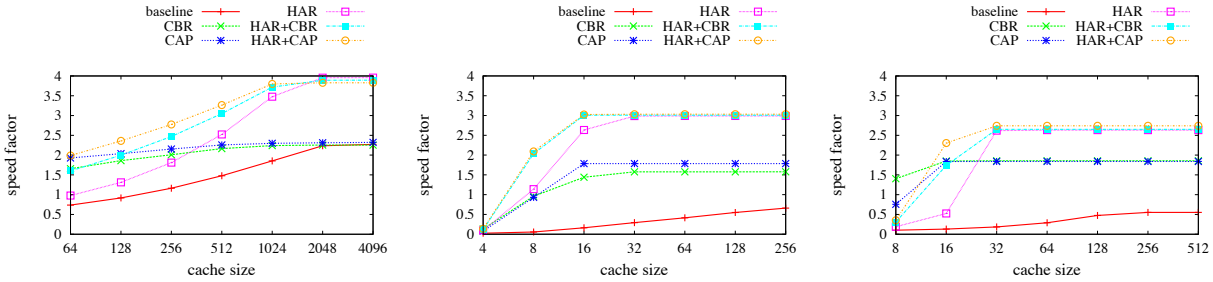
In Synthetic, HAR outperforms CBR and CAP by 1.41X and 1.42X when the cache is no less than 32-container-size. With a small cache (e.g., 8-container-size), CBR and CAP are better. However, because the cache threshold of HAR is small, it is reasonable to allocate sufficient memory for a restore. The hybrid schemes improve restore performance when the cache is small.

The experimental maximum restore performance in each dataset verifies our estimated values in Section 6.3. In summary, we propose to use the hybrid schemes when self-references are common; otherwise the exclusive use of HAR is recommended.



(a) VMDK (b) Linux (c) Synthetic

Figure 7: The comparisons of rewriting algorithms in terms of restore performance. The cache is 512-, 32-, and 64-container-sized in VMDK, Linux, and Synthetic respectively.



(a) VMDK (b) Linux (c) Synthetic

Figure 8: The comparisons of rewriting algorithms under various cache size. Speed factor is the average value of last 20 backups. The cache size is in terms of # of containers.

Table 3: Metadata space overhead of inline reference management approaches. HAR is used in all approaches.

	VMDK	Linux	Synthetic
Reference Counter [24]	4.64MB	328.36KB	6.53MB
GMS [7]	5.26MB	190KB	7.23MB
CMA	58.19KB	2KB	81.62KB

6.6 Garbage Collection

We compare the metadata space overhead among existing inline reference management approaches in Table 3. We assume each reference counter consumes one byte. The metadata overhead of CMA is lowest, and no more than $1/90$ of that of GMS.

We examine how rewriting algorithms affect garbage collection. The number of valid containers after garbage collection exhibits the actual storage cost, and the results are shown in Figure 9. In the initial backups, the baseline has least valid containers, which verifies the discussions in Section 5.2.1. The advantage of HAR becomes more apparent over time, since the proportion of the former part in Equation 1 increases. Finally HAR decreases the number of valid containers by 27.37%, 68.15%, and 68.43% compared to the baseline in VMDK, Linux, and Synthetic respectively. In Synthetic, the number of valid containers increases continuously because the data size increases. The results indicate HAR achieves better storage saving than the baseline, and the merging is no longer necessary in a deduplication system with HAR.

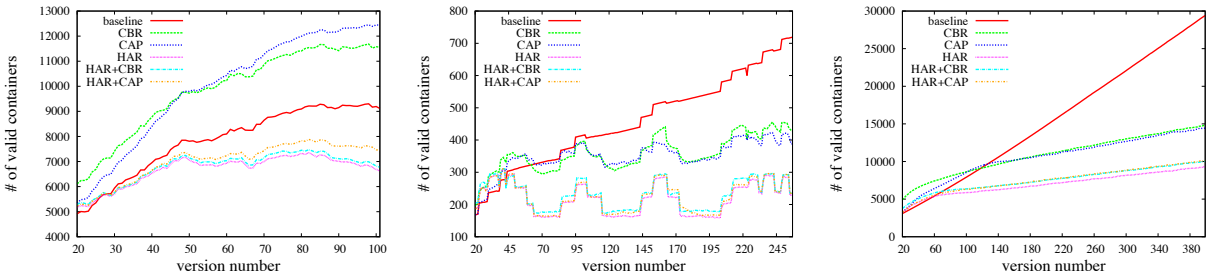
We observe that CBR and CAP increase the number of valid containers by 26.8% and 36.47% respectively in VMDK compared to the baseline. It indicates that CBR and CAP exacerbate the problem of garbage collection in VMDK. The reason is that they rewrite many copies of self-referred chunks into different containers, which reduces the average utilizations as shown in Figure 5. In Linux and Synthetic, CBR and CAP reduce the number of valid containers by 50%, however they still require the merging operation to achieve further storage savings.

HAR+CBR and HAR+CAP respectively result in 2.3% and 12.5% more valid containers than HAR in VMDK. However they significantly reduce the number of valid containers compared with the baseline. They perform slightly worse than HAR in Linux and Synthetic, and outperform CBR and CAP in all three datasets.

6.7 Varying the Utilization Threshold

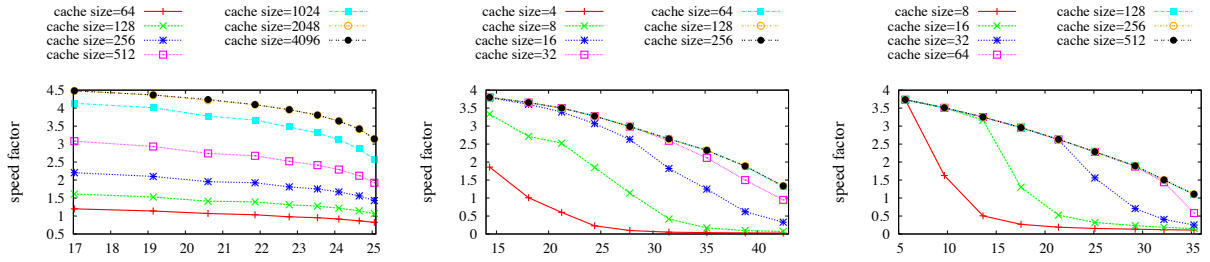
The utilization threshold determines the definition of sparse containers. The impacts of varying the utilization threshold on deduplication ratio and restore performance are both shown in Figure 10.

Varying the utilization threshold from 90% to 10%, the deduplication ratio increases from 17.03 to 25.06 and the restore performance decreases by about 35% in VMDK. In particular, with a 70% utilization threshold and a 2048-container-sized cache, the restore performance exceeds 4 units of speed factor. The reason is that the self-referred chunks restore more data than them-



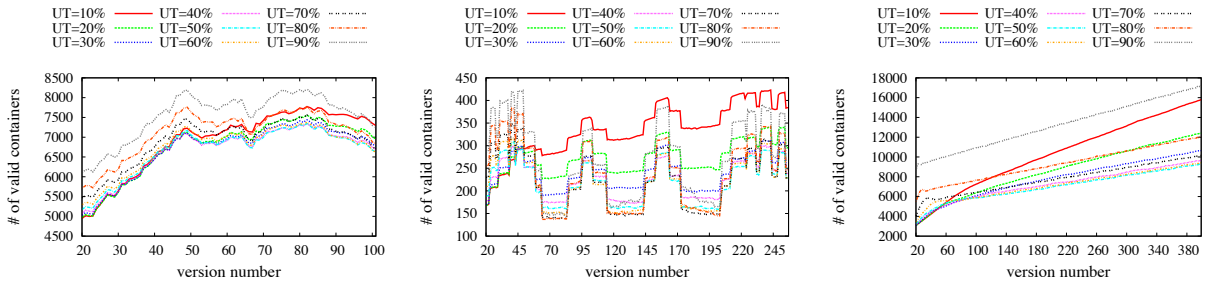
(a) VMDK (b) Linux (c) Synthetic

Figure 9: The comparisons of rewriting algorithms in terms of garbage collection.



(a) VMDK (b) Linux (c) Synthetic

Figure 10: Impacts of varying the utilization threshold on restore performance and deduplication ratio. Speed factor is the average value of last 20 backups. The cache size is in terms of # of containers. Each curve shows varying the utilization threshold from left to right: 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, and 10%.



(a) VMDK (b) Linux (c) Synthetic

Figure 11: Impacts of varying the Utilization Threshold (*UT*) on garbage collection.

selves. In Linux and Synthetic, deduplication ratio and restore performance are more sensitive to the change of the utilization threshold than in VMDK. Varying the utilization threshold from 90% to 10%, the deduplication ratio increases from 14.34 to 42.49, and 5.68 to 35.26 respectively. The smaller the restore cache is, the more significant the performance decrease is as the utilization threshold decreases.

Varying the utilization threshold also has significant impacts on garbage collection. The results are shown in Figure 11. A lower utilization threshold results in less valid containers in initial backups of all our datasets. However, we observe a trend that higher utilization thresholds gradually outperform lower utilization thresholds over time. For instance, the best utilization threshold finally is 50–60% in VMDK, 50–70% in Linux, and 50% in Synthetic. There are some periodical peaks in Linux, since a large upgrade to kernel results in a large amount of emerging sparse containers. These containers will be rewritten in the next backup, which suddenly increases the number of valid containers. After the backup expires, the number of valid containers is reduced.

Based on the experimental results, we believe a 50% threshold is practical in most cases, since it causes moderate rewrites and obtains significant improvements in restore and garbage collection.

7 Conclusions

The fragmentation decreases the efficiencies of restore and garbage collection in deduplication-based backup systems. We observe that the fragmentation comes in two categories: sparse containers and out-of-order containers. Sparse containers determine the maximum restore performance of a backup while out-of-order con-

tainers determine the required cache size to achieve the maximum restore performance.

History-Aware Rewriting algorithm (HAR) accurately identifies and rewrites sparse containers via exploiting historical information. We also implement an optimal restore caching scheme (OPT) and propose a hybrid rewriting algorithm as complements of HAR to reduce the negative impacts of out-of-order containers. HAR, as well as OPT, improves restore performance by 2.6X–17X at an acceptable cost in deduplication ratio. HAR outperforms the state-of-the-art work in terms of both deduplication ratio and restore performance. The hybrid schemes are helpful to further improve restore performance in datasets where out-of-order containers are dominant.

The ability of HAR to reduce sparse containers facilitates the garbage collection. It is no longer necessary to offline merge sparse containers, which relies on identifying valid chunks. We propose a Container-Marker Algorithm (CMA) that identifies valid containers instead of valid chunks. Since the metadata overhead of CMA is bounded by the number of containers, it is more cost-effective than existing reference management approaches whose overhead is bounded by the number of chunks.

Acknowledgments

The work was partly supported by National Basic Research 973 Program of China under Grant No. 2011CB302301; NSFC No. 61025008, 61173043, and 61232004; 863 Project 2013AA013203; Electronic Development fund of Information Industry Ministry. The work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China. The work conducted at VCU was partly supported by US National Science Foundation (NSF) Grants CCF-1102624 and CNS-1218960. The authors are also grateful to Jon Howell and anonymous reviews for their feedback.

References

[1] Linux kernel. http://www.kernel.org/, 2013.

[2] BELADY, L. A. A study of replacement algorithms for a virtual-storage computer. *IBM systems journal 5*, 2 (1966), 78–101.

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

[4] BOTELHO, F. C., SHILANE, P., GARG, N., AND HSU, W. Memory efficient sanitization of a deduplicated storage system. In *Proc. USENIX FAST, 2013*.

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

[6] DUBNICKI, C., GRYZ, L., HELDT, L., KACZMARCZYK, M., KILIAN, W., STRZELCZAK, P., SZCZEPKOWSKI, J., UNGUREANU, C., AND WELNICKI, M. HYDRAstor: A scalable secondary storage. In *Proc. USENIX FAST, 2009*.

[7] GUO, F., AND EFSTATHOPOULOS, P. Building a highperformance deduplication system. In *Proc. USENIX ATC, 2011*.

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

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

[10] 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*.

[11] MEISTER, D., AND BRINKMANN, A. dedupv1: Improving deduplication throughput using solid state drives (SSD). In *Proc. IEEE MSST, 2010*.

[12] MEISTER, D., BRINKMANN, A., AND SÜSS, T. File recipe compression in data deduplication systems. In *Proc. USENIX FAST, 2013*.

[13] MUTHITACHAROEN, A., CHEN, B., AND MAZIÈRES, D. A low-bandwidth network file system. In *Proc. ACM SOSP, 2001*.

[14] 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*.

[15] 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*.

[16] POSEY, B. Deduplication and data lifecycle management. http://searchdatabackup.techtarget.com/tip/Deduplication-and-data-lifecycle-management, 2013.

[17] PRESTON, W. C. *Backup & Recovery*. O'Reilly Media, Inc., 2006.

[18] PRESTON, W. C. Restoring deduped data in deduplication systems. http://searchdatabackup.techtarget.com/feature/Restoring-deduped-data-in-deduplication-systems, 2010.

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

[20] SHILANE, P., HUANG, M., WALLACE, G., AND HSU, W. WAN-optimized replication of backup datasets using stream-informed delta compression. *ACM Transactions on Storage (TOS) 8*, 4 (2012), 13.

[21] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, inline data deduplication for primary storage. In *Proc. USENIX FAST, 2012*.

[22] SYMANTEC. How to force a garbage collection of the deduplication folder. http://www.symantec.com/business/support /index?page=content&id=TECH129151, 2010.

[23] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating realistic datasets for deduplication analysis. In *Proc. USENIX ATC, 2012*.

[24] WEI, J., JIANG, H., ZHOU, K., AND FENG, D. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proc. IEEE MSST, 2010*.

[25] 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*.

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