Reducing Fragmentation for In-line Deduplication Backup Storage via Exploiting Backup History and Cache Knowledge

Min Fu, Dan Feng, *Member, IEEE*, Yu Hua, *Senior Member, IEEE*, Xubin He, *Senior Member, IEEE*, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu

Abstract—In backup systems, the chunks of each backup are physically scattered after deduplication, which causes a challenging fragmentation problem. We observe that the fragmentation comes into sparse and out-of-order containers. The sparse container decreases restore performance and garbage collection efficiency, while the out-of-order container decreases restore performance if the restore cache is small. In order to reduce the fragmentation, we propose History-Aware Rewriting algorithm (HAR) and Cache-Aware Filter (CAF). HAR exploits historical information in backup systems to accurately identify and reduce sparse containers, and CAF exploits restore cache knowledge to identify the out-of-order containers that hurt restore performance. CAF efficiently complements HAR in datasets where out-of-order containers are dominant. 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 2.84-175.36× at a cost of only rewriting 0.5-2.03 percent data.

Index Terms—Data deduplication, storage system, chunk fragmentation, performance evaluation

1 Introduction

Deduplication has become a key component in modern backup systems due to its demonstrated ability of improving storage efficiency [1], [2]. A deduplication-based backup system divides a backup stream into variable-sized chunks [3], and identifies each chunk by its SHA-1 digest [4], 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., 8 KB on average) are managed at a larger unit called *container* [1] that is a fixed-sized (e.g., 4 MB) 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 spatial locality of the backup stream, and a *recipe* is generated to record the fingerprint sequence of the backup. During a restore, the backup stream is reconstructed according to the recipe. The containers serve as the prefetching unit due to the spatial locality. A restore cache holds the prefetched containers and evicts an entire container via an LRU algorithm [5].

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 [6], [7]. The negative impacts of the fragmentation are two-fold. First, the fragmentation severely decreases restore performance [5], [8]. The infrequent restore is important and the main concern from users [9]. Moreover, data replication, which is important for disaster recovery, requires reconstructions of original backup streams from deduplication systems [10], [11], 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 garbage collection solutions first identify valid chunks and the containers holding only a few valid chunks (i.e., reference management [12], [13], [14]). Then, a merging operation is required to copy the valid chunks in the identified containers to new containers [15], [16]. Finally, the identified containers are reclaimed. Unfortunately, the metadata space overhead of reference management is proportional to the number of chunks, and the merging operation is the most time-consuming phase in garbage collection [14].

We observe that the fragmentation comes in two categories of containers: sparse containers and out-of-order containers, which have different negative impacts and require dedicated solutions. 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 intermittently. Both of them hurt the restore performance. Increasing the restore cache size alleviates the negative impacts of out-of-order

- *M. Fu, D. Feng, Y. Hua, J. Liu, W. Xia, F. Huang, and Q. Liu are with the Wuhan National Laboratory for Optoelectronics, School of Computer Science and Technology, Huazhong University of Science and Technology, Division of Data Storage System, Wuhan 430074, China. E-mail: {fumin, dfeng, csyhua, jnliu, xia, huangfangting, qing}@hust.edu.cn.*
- *X. He is with the Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA 23284-3072, USA. E-mail: xhe2@vcu.edu.*
- *Z. Chen is with the National Engineering Research Center for Parallel Computer, Beijing, China. E-mail: chenzuoning@vip.tom.com.*

Manuscript received 25 Sept. 2014; revised 26 Jan. 2015; accepted 2 Mar. 2015. Date of publication 4 Mar. 2015; date of current version 12 Feb. 2016. Recommended for acceptance by Z. Lan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2410781

containers, but it is ineffective for sparse containers because they directly amplify read operations. Additionally, the merging operation is required to reclaim sparse containers in the garbage collection after users delete backups.

Existing solutions [5], [8], [17] propose to rewrite duplicate but fragmented chunks during the backup via *rewriting algorithm*, which is a tradeoff 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 on-going backup stream, and identify the fragmented chunks within the buffer. They fail to accurately 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 remain sparse for the next backup. This observation motivates our work to propose a HAR. During a backup, HAR rewrites the duplicate chunks in the sparse containers identified by the previous backup, and records the emerging sparse containers to rewrite them in the next backup. HAR outperforms existing rewriting algorithms in terms of restore performance and deduplication ratio.

In some cases, such as concurrent restore and datasets like virtual machine images, a sufficient restore cache could be unaffordable. To improve the restore performance under limited restore cache, we develop two optimization approaches for HAR, including an efficient restore caching scheme (OPT) and a hybrid rewriting algorithm. OPT outperforms the traditional LRU since it always evicts the cached containers that will not be accessed for the longest time in the future. The hybrid scheme takes advantages of HAR and existing rewriting algorithms (such as Capping [5]). To avoid a significant decrease of deduplication ratio in the hybrid scheme, we develop a CAF as an optimization of existing rewriting algorithms. CAF simulates the restore cache during a backup to identify the out-of-order containers that are out of the scope of the estimated restore cache. The hybrid scheme significantly improves the restore performance under limited restore cache with a slight decrease of deduplication ratio.

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 CMA that identifies valid containers (holding some valid chunks) instead of valid chunks. Compared to existing reference management approaches, CMA significantly reduces the metadata overhead.

The paper makes the following contributions.

- We classify the fragmentation 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.
- We develop CAF to exploit cache knowledge to identify the out-of-order containers that would hurt restore performance. CAF is used in the hybrid scheme to improve restore performance under limited restore cache without a significant 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 presents 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

The fragmentation problem in deduplication systems has received many attentions. iDedup [18] 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 [7], and a selective deduplication scheme [8] for backup workloads. SAR [19] stores hot chunks in SSD to accelerate reads. RevDedup [20] employs a hybrid inline and out-of-line deduplication scheme to improve restore performance of latest backups. The Context-Based Rewriting algorithm (CBR) [17] and the capping algorithm (Capping) [5] are recently proposed rewriting algorithms to address the fragmentation problem. Both of them buffer a small part of the on-going backup stream during a backup, and identify fragmented chunks within the buffer (generally 10-20 MB). For example, Capping divides the backup stream into fixed-sized segments (e.g., 20 MB), and conjectures the fragmentation within each segment. Capping 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.

Reference management for the garbage collection is complicated since each chunk can be referenced by multiple backups. The offline approaches traverse all fingerprints (including the fingerprint index and recipes) when the system is idle. For example, Botelho et al. [14] build a perfect hash vector as a compact representation of all chunks. Since recipes need to occupy significantly large storage space [21], 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 [13] is expensive and error-prone [12]. Grouped Mark-and-Sweep (GMS) [12] uses a bitmap to mark which chunks in a container are used by a backup.

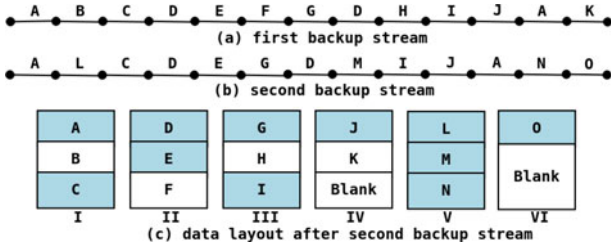


Fig. 1. An example of two consecutive backups. The shaded areas in each container represent the chunks required by the second backup.

3 The Fragmentation Problem

In this section, we describe how the fragmentation arises and how the fragmentation slows down restore and garbage collection. Fig. 1 illustrates an example of two consecutive backups. 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 firstfour containers, and a blank is appended to the 4th half-full container to make it be aligned. With a three-container-sized LRU cache, restoring the first backup needs to read 5 containers. The self-referred chunk A requires extra reading container I.

The second backup is composed of 13 chunks, nine of which are duplicates in the first backup. The four new chunks are stored in two new containers. With a three-container-sized LRU cache, restoring the second backup needs to read nine containers.

Although both of the backups consist of 13 chunks, restoring the second backup needs to read four 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 reported the severe decrease of restore performance in deduplication systems [5], [8], [17]. We observe a $21\times$ decrease in our Kernel dataset (detailed in Section 6.1).

If we delete the first backup, four chunks including chunk K (in container IV) become invalid. However, since they are scattered in four different containers, we cannot reclaim them directly. For example, because chunk J is still referenced by the second backup, we can't reclaim container IV. Existing work uses the offline container merging operation [15], [16]. 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, and becomes the most time-consuming phase in the garbage collection [14].

4 Classification and Observations

In this section, we (1) describe two types of fragmented containers and their impacts, and (2) present our key observations that motivate our work.

4.1 Sparse Container

As shown in Fig. 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 percent, 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 because containers are the prefetching units. Prefetching a container of 50 percent utilization at most achieves 50 percent of the maximum storage bandwidth, because 50 percent of the chunks in the container are never accessed. Hence, the average utilization determines the *maximum restore performance* with an unlimited restore cache. Additionally, 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 is important to 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 reports the probability that all chunks in a container become invalid is low [22]. 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 Fig. 1, container V will be accessed three times intermittently while restoring the second backup. With a three-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. Generally, the datasets with a large amount of self-references suffer more from out-of-order containers.

The impacts of out-of-order containers on restore performance are related to the restore cache. For example, with a four-container-sized LRU cache, restoring the three chunks in container V incurs only one cache miss. With a sufficient restore cache, each related container will only incur 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

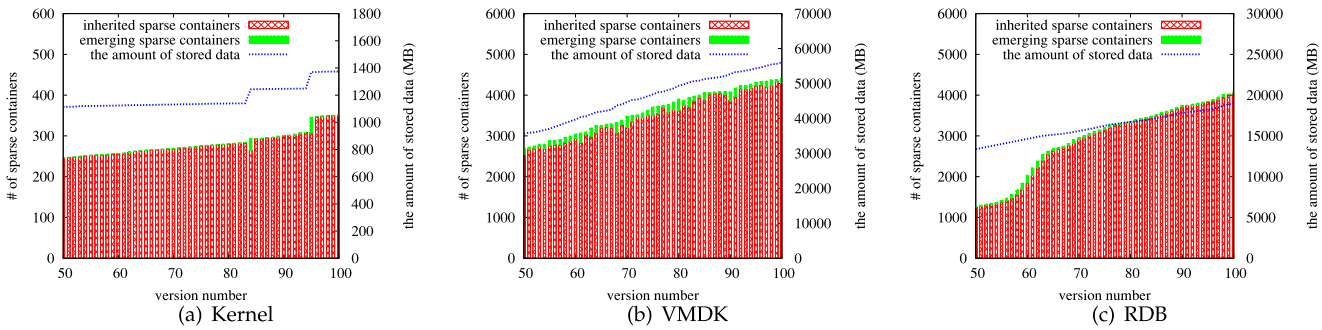


Fig. 2. Characteristics of sparse containers in three datasets. 50 backups are shown for clarity.

memory is expensive, a restore cache of larger than the cache threshold can be unaffordable in practice. For example, there could be multiple concurrent restore procedures, and datasets are dominated by self-references. Hence, it is necessary to either decrease the cache threshold or assure the desired restore performance even 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 when a sufficiently large restore cache is affordable. Existing rewriting algorithms of a limited buffer cannot accurately identify sparse containers, because 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 workloads, two consecutive backups are very similar, which is the major assumption behind DDFS [1]. Hence, they share similar characteristics, including the fragmentation. We analyze three datasets, including Linux kernel (Kernel), a virtual machine (VMDK), and a redis server (RDB) (detailed in Section 6.1), to explore potential characteristics of sparse containers (the utilization threshold is 50 percent). 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 experimental results are shown in Fig. 2. We have three observations. *(1) The number of total sparse containers continuously grows.* It indicates sparse containers become more common over time. *(2) The number of total sparse containers increases smoothly most of time.* It indicates that the number of emerging sparse containers of each backup is relatively small, due to the similarity between consecutive backups. A few exceptions in the Kernel dataset 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. *(3) 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.* It indicates that sparse containers of the backup remain sparse in the next 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 in the current backup.

The above observations motivate our work to exploit the historical information to identify sparse containers, namely History-Aware Rewriting algorithm. 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. Due to the third observation, the scheme identifies sparse containers accurately.

5 Design and Implementation

5.1 Architecture Overview

Fig. 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 Fig. 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.

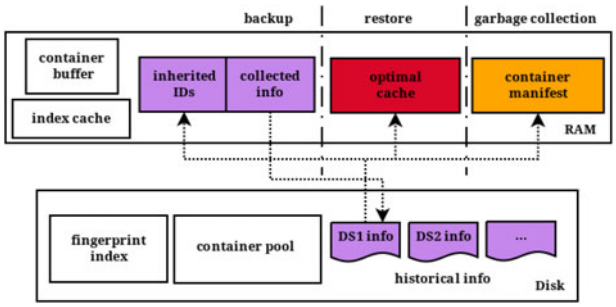


Fig. 3. The HAR architecture. The arrows indicates metadata flows.

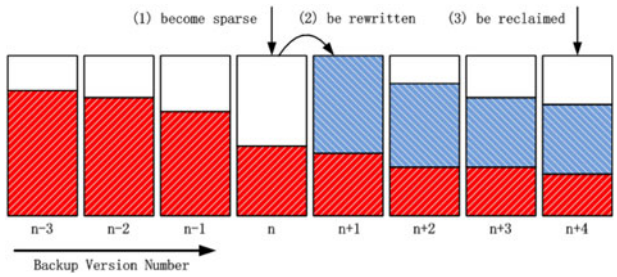


Fig. 4. The lifespan of a rewritten sparse container.

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 (*inherited IDs* in Fig. 3). During the backup, HAR rewrites all duplicate chunks whose container IDs exist in $S_{inherited}$. Additionally, HAR maintains an in-memory structure, $S_{emerging}$ (included in *collected info* in Fig. 3), to monitor the utilizations of all the containers referenced by the backup. $S_{emerging}$ is a set of *utilization records*, and each record consists of a container ID and the current utilization of the container. After the backup concludes, HAR removes the records of higher utilizations than the utilization threshold from $S_{emerging}$. $S_{emerging}$ then contains IDs of all emerging sparse containers.

In most cases, $S_{emerging}$ can be flushed directly to disks as the $S_{inherited}$ of the next backup, because the size of $S_{emerging}$ is generally small due to our second observation. However, there are two spikes in Fig. 2a. A large number of emerging sparse containers indicates that we have many fragmented chunks to be rewritten in next backup. It would change the performance bottleneck to data writing [17] and hurt the backup performance that is of top priority [23]. To address this problem, HAR sets a *rewrite limit*, such as 5 percent, to avoid too much rewrites in next backup.

HAR uses the rewrite limit to determine whether there are too many sparse containers in $S_{emerging}$. **(1)** HAR calculates an estimated *rewrite ratio* (defined as the size of rewritten data divided by the backup size) for the next backup. Specifically, HAR first calculates the estimated size of rewritten chunks for each emerging sparse container via multiplying the utilization by the container size. Then, the estimated rewrite ratio is calculated as the sum of estimated sizes divided by the current backup size, which is approximate to the actual rewrite ratio of the next backup due to the incremental nature of backup workloads. **(2)** If the estimated rewrite ratio exceeds the predefined rewrite limit, HAR removes the record of the largest utilization in $S_{emerging}$ and jump to step 1. **(3)** Otherwise, HAR replaces the IDs of the old inherited sparse containers with the IDs of emerging sparse containers in $S_{emerging}$. The $S_{emerging}$ becomes the $S_{inherited}$ of the next backup. The complete workflow of HAR is described in Algorithm 1.

Fig. 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 latest 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 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.

Algorithm 1. History-Aware Rewriting Algorithm

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

```
Initialize a set, S_emerging.
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 to a new container.
        else
            Eliminate the chunk.
        end if
    else
        Write the chunk to a new container.
    end if
    Update the utilization record in S_emerging.
end while
Remove all utilization records of larger utilizations than the utilization threshold from S_emerging.
Calculate the estimated rewrit ratio for the next backup.
while the estimated rewrite ratio is larger than the rewrite limit do
    Remove the utilization record of the largest utilization in S_emerging.
    Update the estimated rewrite ratio.
end while
return S_emerging
```

Due to the limited number of inherited sparse containers, the memory consumed by the $S_{inherited}$ is negligible. $S_{emerging}$ consumes more memory because it needs to monitor all containers related with the backup. If the default container size is 4 MB and the average utilization is 50 percent which can be easily achieved by HAR, the $S_{emerging}$ of a 1 TB stream consume 8 MB memory (each record contains a 4-byte ID, a 4-byte current utilization, and an 8-byte pointer). The memory footprint is smaller than the rewriting buffer used in CBR [17] and Capping [5].

There is a tradeoff in HAR as we vary the utilization threshold. 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 percent, HAR promises an average utilization of no less than 50 percent, and the maximum restore performance is no less than 50 percent of the maximum storage bandwidth.

5.3 The Impacts of HAR on Garbage Collection

In this section, we discuss how HAR affects the garbage collection efficiency. 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.

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

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

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

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

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

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

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 achieves a lower 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 percent works well according to our experimental results in Section 6.7.

5.4 Optimal Restore Cache

To reduce the negative impacts of out-of-order containers on restore performance, we implement Belady's optimal replacement cache [24]. 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 Fig. 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.

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 percent, the complete sequence of access records of a 1 TB stream consumes over 100 MB 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.

5.5 A Hybrid Scheme

Our observation in Section 6.5 shows that HAR requires a 2048-container-sized restore cache (8 GB) to outperform Capping in a virtual machine image dataset. The memory footprint is around a half image in size. It is because virtual machine images contain many self-references that exacerbate the problem of out-of-order containers. In practice, such a large restore cache could be unaffordable due to concurrent backup/restore procedures.

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 [17] and Capping [5]) as optional optimizations. The hybrid scheme can be straightforward. However, existing rewriting algorithms have no idea of the restore cache, and hence rewrite too many unnecessary out-of-order containers. The simple combination decreases deduplication ratio significantly (generally, its deduplication ratio is smaller than min $(HAR, existing\ rewriting\ algorithms)$). We hence need to optimize existing rewriting algorithms to avoid unnecessary rewrites.

Fig. 5 shows a sample backup stream. The blue container is out-of-order. With a 7-chunk-sized rewriting buffer, the chunk Y would be rewritten by existing rewriting algorithms since the X and Z beyond the scope of the rewriting buffer. However, with a three-container-sized LRU cache, the chunk Y would not hurt the restore performance, since the blue container has been prefetched for restoring X and would remain in the cache while restoring Y. Rewriting Y mistakenly not only reduces storage efficiency but also hurts restore performance.

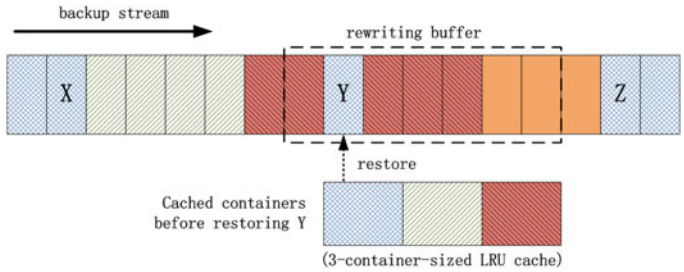


Fig. 5. An example of a backup stream. Chunks with the same pattern are in an identical container. We also show the cached containers before restoring Y.

We develop a CAF to exploit cache knowledge, as an optimization of existing rewriting algorithms. Our key observation is that the sequence of restoring chunks is just the same as the sequence of writing them during the backup. Hence, given an LRU restore cache with a predefined size, we are aware of the runtime state of the restore cache during the backup. CAF simulates the LRU restore cache during the backup using the container IDs of the preceding chunks in the backup stream. For example, when we back up Y, CAF knows the blue container would remain in the restore cache if a three-container-sized LRU cache is used. CAF then denies the request of rewriting Y. CAF improves existing rewriting algorithms in terms of both storage efficiency and restore performance.

CAF needs to estimate the restore cache size, which we call the *estimated cache size*. In practice, CAF may underestimate or over-estimate the restore cache size. 1) *underestimation*. If the restore cache size is larger than the estimated cache size, some chunks that would NOT hurt restore performance are rewritten. However, CAF has avoided a large amount of mistaken rewrites, and thus outperforms existing rewriting algorithms in terms of both deduplication ratio and restore performance. 2) *overestimation*. If the restore cache size is smaller than the estimated cache size, some chunks that would hurt restore performance are NOT rewritten. Hence, CAF improves the deduplication ratio of existing rewriting algorithms at a cost of decreasing restore performance. We suggest to use a large estimated cache size for CAF and satisfy the cache demand during restore if possible. Moreover, the restore cache can employ other replacement algorithms rather than LRU, e.g., the optimal cache proposed in Section 5.4. Based on our experimental results in Section 6.5, CAF works well even if we use the optimal cache.

Fig. 6 shows the possible states (SPARSE, OUT_OF_ORDER, and CACHED) of a duplicate chunk in the hybrid rewriting scheme. The workflow is as follow: (1) The duplicate chunk is checked by HAR. (2) If HAR considers the chunk fragmented, the chunk is marked as SPARSE. Jump to step 9. (3) Otherwise, Capping further checks the chunk (other rewriting algorithms, such as CBR, are also applicable). (4) If Capping considers the chunk NOT fragmented, jump to step 8. (5) Otherwise, the chunk is marked as OUT_OF_ORDER. CAF checks the chunk's container ID in the simulated restore cache. (6) If the chunk is expected in the restore cache, it is marked CACHED. Jump to step 8. (7) Otherwise, jump to 9. (8) The chunk is eliminated. Jump to step 1 for the next chunk. (9) The chunk is rewritten. Jump to step 1 for the next chunk.

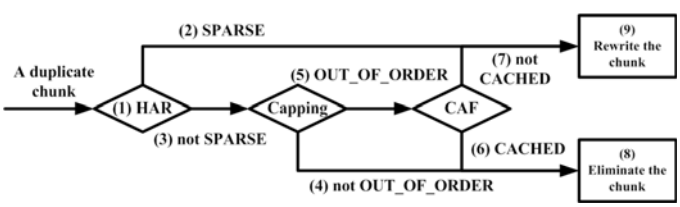


Fig. 6. State transitions of a duplicate chunk in the hybrid rewriting scheme. We use Capping as an example. Other rewriting algorithms are applicable.

Based on our observations in Section 6, only rewriting a small number of additional chunks improves restore performance significantly when the restore cache is small. The hybrid scheme efficiently reduces the cache threshold by a factor of 4 in the virtual machine images. Since the hybrid scheme always rewrites more data than HAR, we suggest to enable the hybrid scheme only in the datasets where self-references are common.

5.6 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 [12], [13], [14] 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 CMA to efficiently determine which containers are invalid. CMA assumes users delete backups in a FIFO scheme, in which oldest backups are deleted first. The FIFO scheme is widely used, such as Dropbox [25] that keeps data of latest 30 days for free users.

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

TABLE 1
Characteristics of Datasets

dataset name	Kernel	VMDK	RDB	Synthetic
total size	104 GB	1.89 TB	1.12 TB	4.5 TB
# of versions	258	126	212	400
deduplication ratio	45.28	27.32	39.11	37.26
avg. chunk size	5.29 KB	5.25 KB	4.5 KB	12.44 KB
sparse	severe	medium	severe	severe
out-of-order	medium	severe	medium	medium

chunk size. After a backup is completed, the backup times of the containers referenced by the backup (HAR already monitors these container IDs) 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). Hence, CMA is fault-tolerant and recoverable.

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 reclaimed.

Because HAR effectively maintains high utilizations of containers, the container manifest is small. We assume that each backup is 1 TB and 90 percent identical to adjacent backups. Recent 20 backups are retained. With a 50 percent average utilization, the backups at most refer to 1.5 million containers. Hence the manifest and lists consume at most 13.5 MB 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 Datasets

Four datasets, including Kernel, VMDK, RDB, and Synthetic, are used for evaluation. Their characteristics are listed in Table 1. Each backup stream is divided into variable-sized chunks via Content-Defined Chunking [3].

Kernel, downloaded from the web[26], is a commonly used public dataset [27]. It consists of 258 consecutive versions of unpacked Linux codes. Each version is 412.78 MB on average. Two consecutive versions are generally 99 percent identical except when there are major revision upgrades. There are only a few self-references and hence sparse containers are dominant.

VMDK is from a virtual machine installed Ubuntu 12.04LTS, which is a common use-case in real-world [12]. We compiled source code, patched the system, and ran an HTTP server on the virtual machine. VMDK consists of 126 full backups. Each full backup is 15.36 GB in size on average, and 90-98 percent identical to its adjacent backups. Each backup contains about 15 percent self-referred chunks. Out-of-order containers are dominant and sparse containers are less severe.

RDB consists of snapshots of a Redis database [28]. The database has 5 million records, 5 GB in space. We ran YCSB [29] to update the database in a Zipfian distribution. The update ratio is of 1 percent on average. After each run, we archived the uncompressed dump.rdb file that is the on-disk snapshot of the database. Finally, we got 212 versions of snapshots. There is no self-reference and hence sparse containers are dominant.

Synthetic was generated according to existing approaches [5], [27]. We simulated common operations of file systems, such as file creation/deletion/modification. We finally obtained a 4.5 TB dataset with 400 versions. There is no self-reference in Synthetic and sparse containers are dominant.

6.2 Experimental Configurations

We implemented our ideas, including HAR, OPT, CAF, and CMA, on Destor that is an open-source platform for data deduplication evaluation [23]. We also implemented Capping [5] for comparisons. Capping is used in the hybrid scheme by default. Since the design of the fingerprint index is out of scope for the paper, we simply accommodate the complete fingerprint index in memory. The *baseline* is of no rewriting, and the default caching scheme is OPT. The container size is 4 MB. The capping level in Capping is 14/20 containers per MB (i.e., a 20 MB rewriting buffer at most refers to 14 containers). The default utilization threshold in HAR is 50 percent. We vary the estimated cache size of CAF according to the characteristics of the datasets, specifically 32-container-size in Kernel, 64-container-size in RDB and Synthetic, and 512-container-size in VMDK. We expect a larger cache in VMDK since out-of-order containers are dominant. CAF is used to refer to CAF-optimized Capping in the following. We retain the latest 30 backups at any moment. We don't apply the offline container merging as in previous work [5], [8], because it requires a long idle time.

We use Restore Speed Factor [5] as the metric of the restore performance. The restore speed factor is defined as 1 divided by mean containers read per MB of restored data. It indicates how much data restored per container read. A higher restore speed factor indicates better restore performance because we reconstruct a backup stream via less container reads. Given the container size is 4 MB, 4 units of restore speed factor correspond to the maximum storage bandwidth.

6.3 Average Utilization

The average utilization of a backup exhibits its theoretically maximum restore performance with unlimited restore cache. We calculate the utilization via dividing the backup size by the total size of actually referred containers. Fig. 7 shows the average utilizations of several rewriting algorithms. The baseline has lowest average utilizations in all datasets due to its severe fragmentation. Capping improves the average utilization by 1.24-3.1× than the baseline. CAF has similar average utilizations to Capping although CAF rewrites less data. We observe that HAR as well as the hybrid scheme obtain highest average utilizations in all datasets. The average utilizations of HAR are

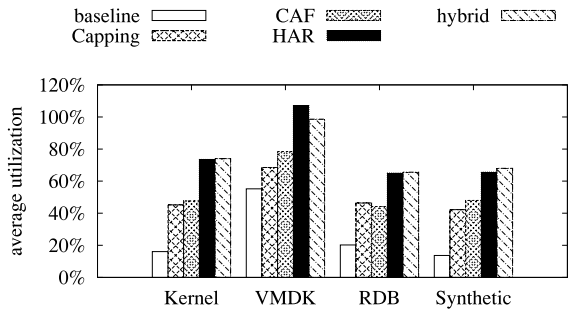


Fig. 7. The average utilization of last 20 backups achieved by each rewriting algorithm.

73.55, 107.09, 65.13, and 65.6 percent in Kernel, VMDK, RDB, and Synthetic respectively, which indicate the *maximum restore speed factors* ($= average\ utilization * 4$) are 2.94, 4.28, 2.61, and 2.62. In VMDK, the average utilization of HAR exceeds 100 percent since VMDK has many self-references that can restore more data than themselves (a self-referred chunk can restore more data than its size since it is restored multiple times in a restore).

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 in a FIFO scheme to trigger garbage collection, the actual storage cost after garbage collection is shown in Section 6.6.

Fig. 8 shows deduplication ratios of rewriting algorithms. The deduplication ratios of HAR are 27.43, 23.96, 24.6, and 21.2 in Kernel, VMDK, RDB, and Synthetic respectively. HAR writes 65.25, 13.96, 58.99, and 75.73 percent more data than the baseline respectively. The average rewrite ratios remain at a low level, respectively 1.46, 0.5, 1.51, and 2.03 percent. It indicates the size of rewritten 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 of backups. Hence, HAR has trivial impacts on the backup performance.

We observe that HAR achieves considerably higher deduplication ratios than Capping. The rewrite ratios of Capping are around two times larger than that of HAR. With the help of CAF, Capping achieves comparable deduplication ratio to HAR. The hybrid scheme achieves better deduplication ratio than Capping, but decreases deduplication ratios compared with HAR, such as by 16.3 percent in VMDK.

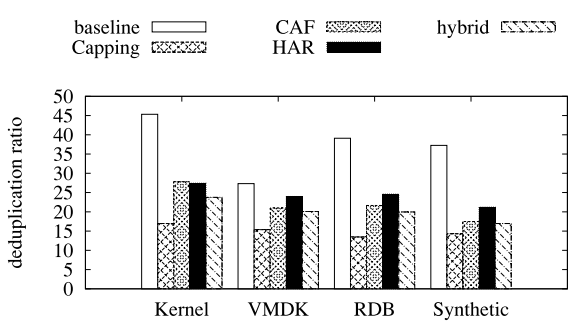


Fig. 8. The comparisons between HAR and other rewriting algorithms in terms of deduplication ratio.

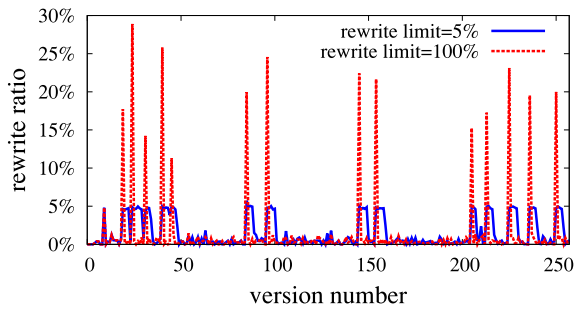


Fig. 9. The impacts of varying the rewrite limit of HAR in Kernel.

Fig. 9 shows how the rewrite limit of HAR works. We compare two different values of the rewrite limit, 5 and 100 percent. Only the results of Kernel are given because there is no sharp increase of sparse containers in other datasets as shown in Fig. 2. If the rewrite limit is 100 percent, HAR rewrites all referenced chunks in sparse containers and hence results in high rewrite ratios in a few backups. For example, the backup 24 has a 28.9 percent rewrite ratio. The occasional high rewrite ratios hurt the backup performance of these backups [17]. We observe that a rewrite limit of 5 percent successfully limits the upper bound of rewrite ratio. The fragmented chunks of the backup 24 are amortized by the following 5 backups. Hence, HAR would not hurt backup performance even when bursting sparse containers.

6.5 Restore Performance

In this section, we compare the rewriting algorithms in terms of restore performance. Due to the limited space, the experimental results from the Synthetic dataset are not shown, which are similar with those from RDB.

Fig. 10 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 Fig. 11. The default caching scheme is OPT. We observe severe declines of the restore performance in the baseline. For instance, restoring the latest backup is 21× slower than restoring the first backup in Kernel. OPT alone increases restore performance by a factor of 1.5-2, however the performance remains at a low level.

We further examine the restore performance of the rewriting algorithms. In Kernel, the restore performance of Capping in last backups is around 2 units of restore speed factor. Compared to Capping, CAF achieves comparable restore performance although it rewrites less data. HAR is better than Capping and CAF in terms of restore performance. HAR as well as OPT improve the restore performance of the baseline by a factor of 14.43. There are some occasional smaller values in the curve of HAR. That is because a large upgrade in Linux kernel produces a large amount of sparse containers. The fragmented chunks in the sparse containers are amortized by following backups due to the rewrite limit of 5 percent. The hybrid scheme achieves best restore performance.

In VMDK, out-of-order containers are dominant and sparse containers are less severe. The restore performance of Capping in last backups is around 2.5-3 units of restore speed factor. CAF improves the restore performance of Capping by a factor of around 1.14, since it avoids a large amount of mistaken rewrites. HAR as well as OPT improve

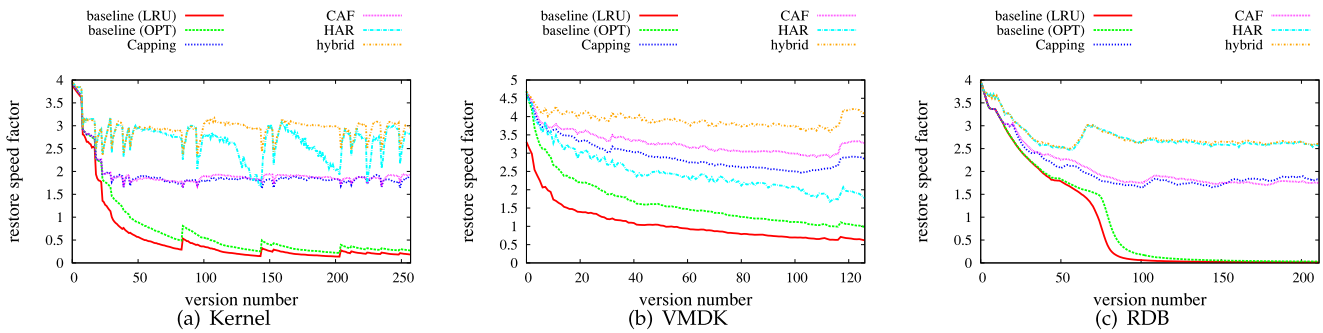


Fig. 10. The comparisons of rewriting algorithms in terms of restore performance. The cache is 32-, 512-, and 64-container-sized in Kernel, VMDK, and RDB respectively.

the restore performance of the baseline by a factor of 2.84, which is worse than Capping since out-of-order containers are dominant in VMDK. But we will see later that HAR is able to achieve best restore performance with a larger restore cache. The hybrid scheme is significantly better than other rewriting algorithms since it takes advantages of HAR, Capping, and CAF. The restore performance of the initial backups exceeds the maximum storage bandwidth (4 units of restore speed factor) due to the large amount of self-references in VMDK.

The results in RDB are similar with those in Kernel. CAF has comparable restore performance to Capping while rewrites less data. HAR achieves best restore performance, 175.36× higher than the baseline. The hybrid scheme can't outperform HAR remarkably since sparse containers are dominant in RDB.

Fig. 11 compares restore performance among rewriting algorithms under various cache sizes. For CAF and the hybrid scheme, we use a constant estimated restore cache in each dataset no matter how the restore cache varies. It is because we cannot exactly predict the available memory in advance. In all datasets, HAR performs worse than Capping in a small cache, but better in a large cache. It is because when the cache is small, out-of-order containers are dominant and thus HAR that focuses on sparse containers underperforms. When the cache is large, out-of-order containers never hurt restore performance and the restore performance is determined by the average utilizations as shown in Fig. 7. As a result, HAR performs best if we have a large restore cache. In backup systems, a large restore cache is common since restore is rare and critical.

We observe that the hybrid scheme improves the restore performance of HAR when the restore cache is small without decreasing restore performance when the restore cache is large. For example, in VMDK, the restore performance of the hybrid scheme is approximate to that of Capping when the cache is small, while it is approximate to that of HAR when the cache is large. The hybrid scheme reduces the cache threshold of HAR in all datasets. Taking VMDK as an example, the cache threshold of HAR is 2,048-container-size, and the hybrid scheme reduces the cache threshold by 4×. The cache thresholds of HAR in Kernel and RDB are small, therefore a restore cache of reasonable size can address the problem caused by out-of-order containers without decreasing deduplication ratio. We suggest to disable the hybrid scheme in Kernel and RDB for storage saving, while enable the hybrid scheme in VMDK which has a significantly larger cache threshold due to self-references.

It is interesting that CAF also underperforms when the restore cache is small. The reasons are two-fold. First, a smaller restore cache than the estimated cache in CAF (the over-estimation case discussed in Section 5.5) would degrade the restore performance of Capping. Second, when the restore cache size drops to the capping level in Capping, CAF becomes inefficient. One assumption of CAF is that we have a much larger restore cache than the rewriting buffer in Capping.

6.6 Garbage Collection

We examine how rewriting algorithms affect garbage collection in this section. The number of valid containers after garbage collection exhibits the actual storage cost, and all invalid containers are immediately reclaimed. The results are shown in Fig. 12. In the initial backups, the baseline has least valid containers, which verifies the discussions in

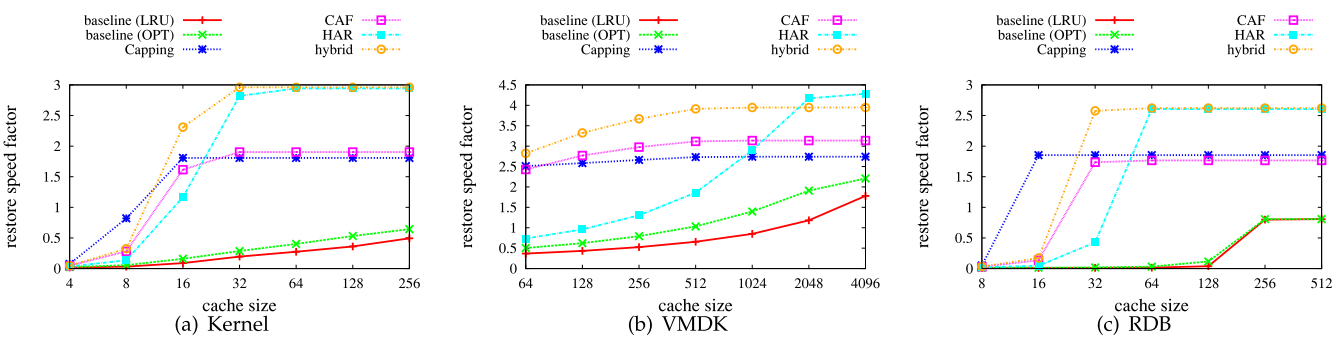


Fig. 11. The comparisons of rewriting algorithms under various cache size. Restore speed factor is the average value of last 20 backups. The cache size is in terms of # of containers.

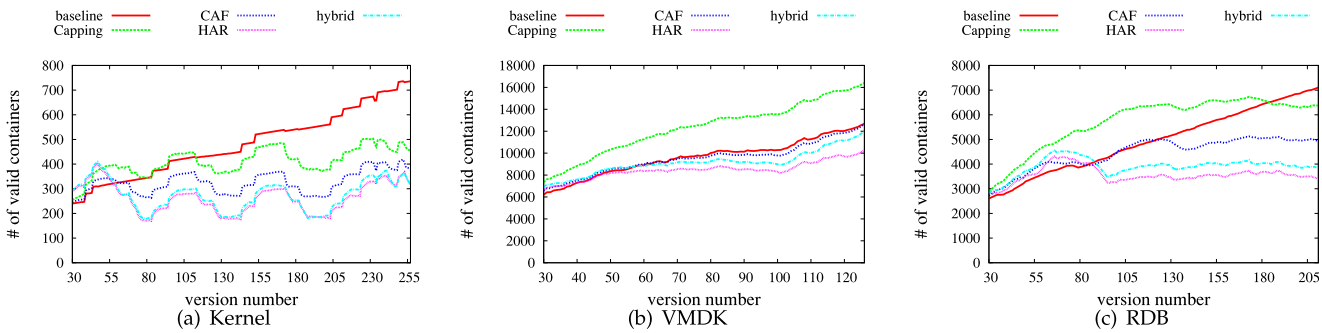


Fig. 12. The comparisons of rewriting algorithms in terms of storage cost after garbage collection.

Section 5.3. The advantage of HAR becomes more obvious over time, since the proportion of the former part in Equation (1) increases. In last 20 backups, HAR decreases the number of valid containers by 52.48, 19.35, and 49 percent compared to the baseline in Kernel, VMDK, and RDB respectively. The results indicate HAR achieves better storage saving than the baseline after garbage collection, and the container merging opearion is no longer necessary in a deduplication system with HAR.

We observe that Capping increases the number of valid containers by 1.3× in VMDK compared to the baseline. It indicates that Capping exacerbates the problem of garbage collection in VMDK. In other words, although Capping results in each backup version refers to less containers than the baseline, all 30 latest live backup versions together refer to more containers. The reason is that Capping rewrites many copies of self-referred chunks into different containers, which reduces the average utilizations. Capping also increases the number of valid containers in the early 180 backups of RDB which has no self-reference. An intuitive explanation is that Capping generally rewrites a part of chunks in a sparse container that makes the rewritten chunks in new containers and the old sparse container still being referenced. In last 20 backups of Kernel and RDB, Capping reduces the number of valid containers by 34 and 7.84 percent respectively.

After avoiding many unnecessary rewrites, CAF achieves lower storage costs than Capping after garbage collection in all datasets. For example, in VMDK, CAF reduces the number of valid containers by 25 percent than Capping. The hybrid scheme significantly reduces the number of valid containers compared with the baseline, while results in slightly more valid containers than HAR. It outperforms Capping and CAF in all 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 Fig. 13.

In VMDK, as the utilization threshold varies from 90 to 10 percent, the deduplication ratio increases from 17.3 to 26.84 and the restore performance decreases by about 38 percent. In particular, with a 70 percent utilization threshold and a 2,048-container-sized cache, the restore performance exceeds 4 units of restore speed factor. The reason is that the self-referred chunks restore more data than themselves. In Kernel and RDB, deduplication ratio and restore performance are more sensitive to the change of the utilization threshold than in VMDK, since sparse containers are dominant in Kernel and RDB. As the utilization threshold varies from 90 to 10 percent, the deduplication ratio increases from 17.06 to 42.44, and from 13.84 to 35.37 respectively. The deduplication ratio has not declined significantly as the utilization threshold varies from 80 to 90 percent in RDB, due to the rewrite limit of 5 percent. 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, as shown in Fig. 14. 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

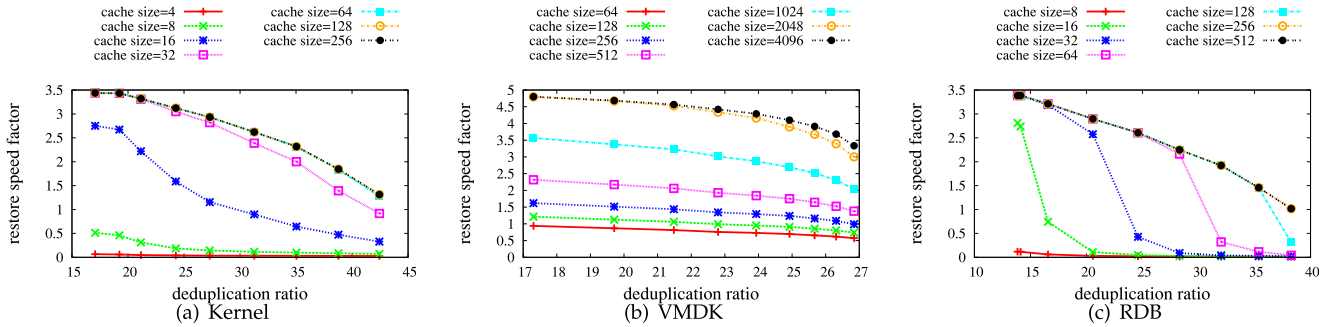


Fig. 13. Impacts of varying the utilization threshold on restore performance and deduplication ratio. Restore speed factor is the average value of last 20 backups. The cache size is in terms of # of containers. Each curve shows varying the utilization threshold from left to right: 90%, 80%, 70%, 60%, 50%, 40%, 30%, 20%, and 10%.

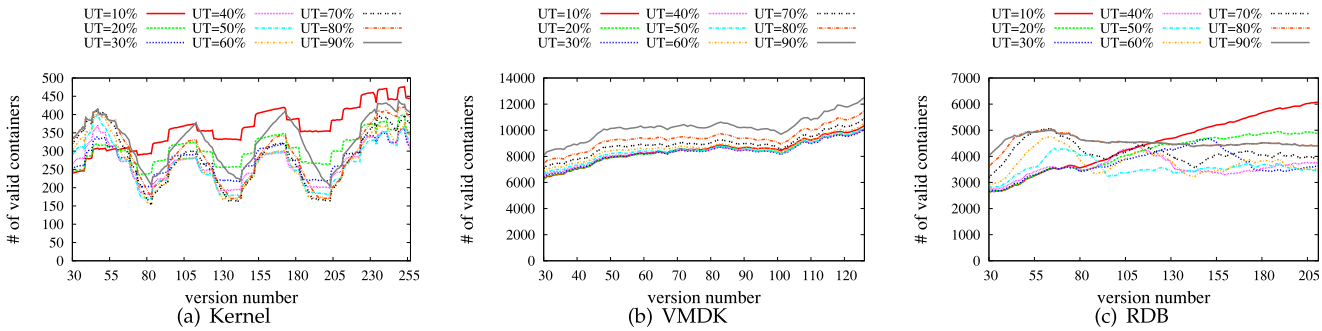


Fig. 14. Impacts of varying the Utilization Threshold (*UT*) on storage cost after garbage collection.

lower utilization thresholds over time. The best utilization thresholds finally are 30-50 percent in Kernel, 20-40 percent in VMDK, and 50-60 percent in RDB. There are some periodical peaks in Kernel, since a large upgrade to Linux kernel results in a large amount of emerging sparse containers. These containers will be rewritten in the following backups, 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 percent utilization threshold is practical in most cases, since it causes moderate rewrites and obtains significant improvements in terms of restore performance and garbage collection efficiency.

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, while out-of-order containers determine the restore performance under limited restore cache.

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 2.84-175.36× 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 scheme is helpful to further improve restore performance in datasets where out-of-order containers are dominant. To avoid a significant decrease of deduplication ratio in the hybrid scheme, we develop a CAF to exploit cache knowledge. With the help of CAF, the hybrid scheme significantly improves the deduplication ratio without decreasing the restore performance. Note that CAF can be used as an optimization of existing rewriting algorithms.

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 chunk-level reference management to identify valid chunks. We propose a 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.

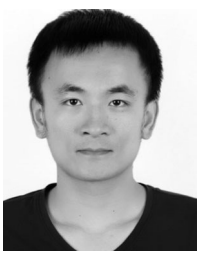
Acknowledgments

This 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; Fundamental Research Funds for the Central Universities, HUST, under Grant No. 2014QNRC019. The work conducted at VCU was partially sponsored by US National Science Foundation under Grants CNS-1320349 and CNS-1218960. The work was also supported by Key Laboratory of Information Storage System, Ministry of Education, China. The preliminary manuscript is published in the proceedings of USENIX Annual Technical Conference (ATC), 2014. J. Liu is the corresponding author.

References

[1] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system,' in *Proc. USENIX FAST*, 2008.
[2] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "HYDRAstor: A scalable secondary storage." in *Proc. USENIX FAST*, 2009.
[3] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proc. ACM SOSP*, 2001.
[4] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proc. USENIX FAST*, 2002.
[5] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. USENIX FAST*, 2013.
[6] (2010). Restoring deduped data in deduplication systems. [Online]. Available: http://searchdatabackup.techtarget.com/feature/Restoring-deduped-data-in-deduplication-systems
[7] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *Proc. IEEE High Performance Comput. Commun.*, 2011, pp. 581–586.
[8] Y. J. Nam, D. Park, and D. H. Du, "Assuring demanded read performance of data deduplication storage with backup datasets," in *Proc. IEEE MASCOTS*, 2012, pp. 201–208.
[9] W. C. Preston, *Backup & Recovery*. O'Reilly Media, Inc., 2006.
[10] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN-optimized replication of backup datasets using stream-informed delta compression," in *Proc. USENIX FAST*, 2012.
[11] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: Coarse-grained data reordering to improve compressibility," in *Proc. USENIX FAST*, 2014.
[12] F. Guo and P. Efstathopoulos, "Building a highperformance deduplication system," in *Proc. USENIX ATC*, 2011.
[13] J. Wei, H. Jiang, K. Zhou, and D. Feng, "MAD2: A scalable high-throughput exact deduplication approach for network backup services," in *Proc. IEEE Mass Storage Syst. Technol.*, 2010, pp. 1–14.
[14] F. C. Botelho, P. Shilane, N. Garg, and W. Hsu, "Memory efficient sanitization of a deduplicated storage system," in *Proc. USENIX FAST*, 2013.

[15] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. USENIX FAST*, 2009.
[16] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives (SSD)," in *Proc. IEEE Mass Storage Syst. Technol.*, 2010, pp. 1–6.
[17] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *Proc. ACM SYSTOR*, 2012.
[18] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proc. USENIX FAST*, 2012.
[19] B. Mao, H. Jiang, S. Wu, Y. Fu, and L. Tian, "Read-performance optimization for deduplication-based storage systems in the cloud", *ACM Trans. Storage*, vol. 10, no. 2, pp. 6:1–6:22, Mar. 2014.
[20] Y.-K. Li, M. Xu, C.-H. Ng, and P. P. C. Lee, "Efficient hybrid inline and out-of-line deduplication for backup storage", *ACM Trans. Storage*, vol. 11, no. 1, pp. 2:1–2:21, Dec. 2014.
[21] D. Meister, A. Brinkmann, and T. Süß, "File recipe compression in data deduplication systems," in *Proc. USENIX FAST*, 2013.
[22] (2010). How to force a garbage collection of the deduplication folder. [Online]. Available: http://symantec.com/business/support/index?page=//content/&id=TECH129151
[23] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication tradeoffs in backup workloads," in *Proc. USENIX FAST*, 2015.
[24] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer", *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.
[25] (2015). Dropbox. [Online]. Available: https://ropbox.com/
[26] (2015). Linux kernel. [Online]. Available: http://kernel.org/
[27] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. USENIX ATC*, 2012.
[28] (2015). Redis. [Online]. Available: http://redis.io/
[29] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb", in *Proc. ACM SoCC*, 2010.



Min Fu is currently working toward the PhD degree majoring in computer architecture in Huazhong University of Science and Technology, Wuhan, China. His current research interests include data deduplication, backup and archival storage system, and key-value store. He publishes several papers in major conferences including USENIX ATC, FAST, etc.

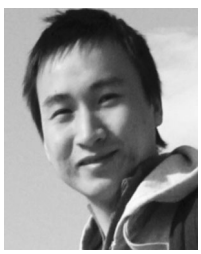
Dan Feng received the BE, ME, and PhD degrees in computer science and technology in 1991, 1994, and 1997, respectively, from Huazhong University of Science and Technology (HUST), Wuhan, China. She is currently a professor and vice dean of the School of Computer Science and Technology, HUST. Her research interests include computer architecture, massive storage systems, and parallel file systems. She has more than 100 publications in major journals and international conferences, including *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, ACM-TOS, JCST, FAST, USENIX ATC, ICDCS, HPDC, SC, ICS, IPDPS, and ICPP. She serves on the program committees of multiple international conferences, including SC 2011, 2013 and MSST 2012. She is a member of IEEE and a member of ACM.

Yu Hua received the BE and PhD degrees in computer science from the Wuhan University, Wuhan, China, in 2001 and 2005, respectively. He is currently an associate professor at the Huazhong University of Science and Technology, Wuhan, China. His research interests include computer architecture, cloud computing, and network storage. He has more than 60 papers to his credit in major journals and international conferences including *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, USENIX ATC, USENIX FAST, INFOCOM, SC, ICDCS, ICPP. He has been on the program committees of multiple international conferences, including INFOCOM, ICDCS, ICPP and IWQoS. He is a senior member of the IEEE, and a member of ACM and USENIX.

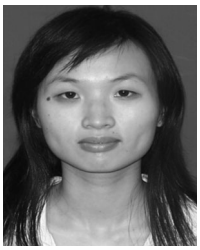
Xubin He received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1997, respectively, and the PhD degree in electrical engineering from University of Rhode Island, Kingston, RI, in 2002. He is currently a professor in the Department of Electrical and Computer Engineering at Virginia Commonwealth University, Richmond, VA. His research interests include computer architecture, storage systems, virtualization, and high availability computing. Dr. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the Sigma Xi Research Award (TTU Chapter) in 2005 and 2010. He is a senior member of the IEEE, a member of the IEEE Computer Society and USENIX.

Zuoning Chen received the ME degree in Computer Science and Technology from Zhejiang University, Hangzhou, China. She is a fellow of Chinese Academy of Engineering, and engaged in research of system software and computer architecture. Her current research interests include secure platform for domestic CPU, and big data.

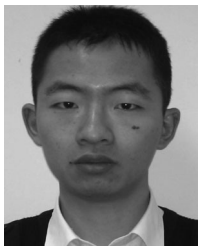
Jingning Liu received the BE degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1982. She is a professor in the HUST and engaged in researching and teaching of computer system architecture. Her research interests include computer storage network system, high-speed interface and channel technology, embedded system.



Wen Xia received the PhD degree in computer science from Huazhong University of Science and Technology (HUST), Wuhan, China, in 2014. He is currently an assistant professor in the school of computer science and technology at HUST. His research interests include data deduplication, delta compression, and storage systems. He publishes more than 10 papers in major journals and international conferences including *IEEE Transactions on Computers*, USENIX ATC, USENIX FAST, INFOCOM, IFIP Performance, IEEE DCC, etc.



Fangting Huang received the BE degree in software engineering from the Sun Yetsen University (SYSU), Guangzhou, China, in 2010. She is currently working toward the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China. Her research interest includes computer architecture and storage systems.



Qing Liu received the BE degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2009. He is currently working toward the PhD degree in school of computer science and technology in HUST. His interests include erasure codes, storage systems, and distributed storage systems.