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

By 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

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

Presented by Ariel Kolikant and Arthur Sapozhnikov
The story so far...

- No doubt Deduplication improves storage efficiency
- Deduplication has so drastically improved storage that it would be impossible to ever revert back
- We have seen multiple ways of improving on it and have achieved magnificent efficiency thanks to deduplication

.....so what’s the problem?
Reminder of how our deduplication works:

- Containers are basic units for writing operations.
- During backup, chunks are aggregated into containers.
- During a Restore, a recipe (fingerprint sequence of backup) is used and the containers are PREFETCHED into an LRU cache.
- When containers are filled, they remain closed until all chunks are invalidated.
- When a duplicate chunk is to be backed, we refrain from adding it, since it is already saved.
Wait.....

- So duplicate chunks are eliminated between multiple backups?
  - So you are saying that files are split across multiple containers??
- Containers are locked until they are completely invalidated?
  - So you are saying that we have multiple invalid chunks stored??
Wait.....

- So duplicate chunks are eliminated between multiple backups?
- So you are saying that files are split across multiple containers??
- They are totally invalid?!
- The multiple invalid chunks stored??

Deduplication Causes Fragmentation!
The Fragmentation Problem: Backup chunks are physically scattered across multiple containers

- Severely decrease restore performance!
- Data replication (for disaster recovery) needs to reconstruct the original streams, thus suffers the same problems as restore!
- Invalid chunks are scattered amongst multiple containers! Making garbage collection difficult.
- Existing solutions to the Garbage collection problem identify valid chunks in containers that are mostly dead and merge them. This merge operation becomes the most time consuming phase of garbage collection!

Let’s see an example to this behavior!
Self reference

3-container-sized LRU Cache
Conclusions 1

- 5 container read for a 5 container sized stream? That’s wonderful! So what’s the problem?
  - The first backup was UNFRAGMENTED, the entire stream was saved as compatible and serial as possible.
  - Let’s take a look at the second backup....
Animations were too complicated, so this will be done on blackboard.

**Notice to move LRU accordingly to the last accessed -chunk--**
Conclusions 2

- 9 accesses for 5 container sized stream after only a single additional back up.
- This was a small example it is easy to see how this problem explodes in size

- Fragmentation damages restore time and is a serious issue for all the reasons already stated.
- We need a way to maintain deduplication, while eliminating fragmentation
So far, what we need to understand is that fragmentation is a problem, and the algorithm that this article offers is the **HAR** algorithm in the hopes of solving it.

Here is a diagram of what you are expected to know so far.

**HAR**: The History aware Rewriting Algorithm
Fragmentation types

An observation made in the paper splits fragmented containers into two categories:

- **Sparse Containers**: containers containing only a few valid chunks
  - This reduces both restore performance and garbage collection efficiency
- **Out-Of-Order Containers**: chunks are accessed intermediately
  - This reduces restore performance, but can be improved by increasing the cache size
Sparse Containers

- **Utilization rate:** Every container might hold both valid and invalid chunks. The container’s utilization rate is defined as $\frac{\text{# valid chunks}}{\text{# total chunks}}$. The backup’s utilization rate is defined as the average utilization rate of all of its valid containers.

- **Sparse container:** A container which has a utilization rate lower than a predefined utilization threshold (e.g. 50%).

- Invalid chunks in sparse containers are not reclaimed until all chunks are invalidated. Of which probability is low.

- Merging, which occurs in most solutions to the problem, suffers from performance operations...
Out-Of-Order Containers

- If a container is accessed many times intermittently during a restore, we consider it as an *out-of-order container for the restore*. Notice that the performance issues caused by these containers happen only if caches are insufficiently large.

- We define Cache Threshold to be the minimum cache size required for the maximum restore performance (under average utilization).
  - So, if the cache is smaller than the cache threshold, performance is damaged, if they are larger it is unaffected.
  - But since that isn’t always possible, other optimizations are possible: decreasing the cache threshold (by changing stream), moving chunks that are called for together, to the same container, more intelligent caching schemes than LRU.

- This article mostly focuses on Sparse containers, but would tackle OOOC on several occasions.
The Key Observation

- 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!
- Therefor we can use a History-Aware-Algorithm in order to use previous backup information for bettering next backups!
- Let us prove these quite huge assumptions first...
Inherited sparse containers: containers that were sparse in the last backup and would be sparse in this backup.

Emerging sparse containers: containers that were not sparse in the last backup, but are in the current backup.

1) The number of total sparse containers continuously grows.
2) Second, the number of total sparse containers increases smoothly most of the time.
3) 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.

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

- **During back up**
  - HAR rewrites the duplicate chunks that are in a sparse container, identified by the last backup. Using information about the last backup, saved in the history.
  - records the emerging sparse containers to rewrite them in the next backup

- **During garbage collection**
  - Before HAR, gc would have to identified valid chunks and merge them, which is cumbersome and error prone
  - But, since HAR efficiently reduces sparse containers, the identification of valid chunks is no longer necessary!

- **Assumption**
  - We save the historic information of the \(-t\)- previous backups which are enough to better current backups
Har Algorithm: chunk part

Input: IDs of inherited sparse containers, \textit{Sinherited} ;  
Output: IDs of emerging sparse containers, \textit{Ssparse};  
1: Initialize two sets, \textit{Ssparse} and \textit{Sdense}.  
2: while the backup is not completed do  
3: Receive a chunk and look up its fingerprint in the fingerprint index.  
4: if the chunk is duplicate then  
5: if the chunk’s container ID exists in \textit{Sinherited} then  
6: Rewrite the chunk, and obtain a new container ID.  
7: else  
8: Eliminate the chunk.  
9: end if  
10: else  
11: Write the chunk, and obtain a new container ID.  
12: end if
Har Algorithm: container part

if the chunk’s container ID doesn’t exist in $S_{dense}$
then
14: Update the associated utilization record (add it if doesn’t exist) in $S_{sparse}$ with the chunk size.
15: if the utilization exceeds the utilization threshold
then
16: Move the utilization record to $S_{dense}$.
17: end if
18: end if
19: end while
20: return $S_{sparse}$
The Impacts of HAR on Garbage Collection

- In the paper it has been proven that HAR has positive implications on GB if the number of backups is large and negative implications if the number of backups is small.
  - That is due to the fact that it does improve performance but carries an overhead that is only surpassed if otherwise a lot of information would have been saved.
To reduce the negative impacts of out-of-order containers on restore performance, we implement Belady’s optimal replacement cache.

- By using an “access record” saved in the “collected info”
- Access record, is a record of a backup stream’s container accesses

During backup save access record of backup’s containers.

During restore, prefetch by the access record saved during backup, thus reduce the number of OOOC of a stream’s restore.
5.1 Architecture Overview

Figure 3: The HAR architecture.
HAR, conclusions

- Two fragmentation types. One of them due to cache size one of them due to low utilization
  - OOOC can be solved with bigger cache
- Both fragmentation types can be reduced in quantity by using historical information
  - Sparse due to the fact that sparse containers are inherited in the next backup
  - OOOC due to remembering access records and prefetching the required containers
A Hybrid Scheme

- CBR, CAP: algorithms that are used to reduce the amount of oooc
  - Every chunk not rewritten by HAR is looked at by either of those algorithms. If they wish to rewrite it, they may.
  - We use historical information to avoid OOOC containers, however these algorithms use other methods.
  - In order to not harm deduplication too much. A limit is set on the percentage of rewrites these algorithms can perform

- The Hybrid scheme always rewrite more chunks than HAR, therefore it is recommended to disable it in case of a large restore cache (since you can afford to have many OOOC)
CMA: 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 and other approaches 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, a bit of how it’s done

- **CMA** maintains a *container manifest* for each backup.
  - 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.
  - For each container, CMA maintains a dataset list that records IDs of the datasets referring to the container.
    - If a containers list is empty, we can reclaim the container!
  - So, we can do GC on CONTAINER resolution, if we delete the last backup to touch a container, GC can invalidate the entire container.
CMA, a bit of how it’s done

- CMA maintains a *container manifest* for each backup.
  - 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.
  - For each container, CMA maintains a dataset list that records IDs of the datasets referring to the container.
  - If a containers list is empty, we can reclaim the container.

- So, we can do GC on CONTAINER resolution, if we delete the last backup to touch a container, GC can invalidate the entire container.
Don’t worry Fry!

- HAR maintains high utilization (by removing sparse containers)
  - So for a series of backups sized 1TB each with 90% identicality to the previous backups, with retaining 20 backups back and a 50% utilization and 1.5 million referred containers....
  - The storage space needed for the container manifest and list consume at most 13.5 MB which compared to even just 1 dataset of 1TB is still peanuts.
And Now, Performance Evaluation
The configurations....

- The algorithms being compared are:
  - HAR
  - CBR, CAP
  - Hybrid schemes: HAR+CBR, HAR+CAP.
  - Baseline (which is without any fancy algorithms)

- The default caching is OPT (optimal restore caching scheme)
- Container size of 4MB
- For HAR:
  - Default utilization threshold of 50%
  - Retain 20 backups in HAR (deleting backup n-20 after backup n)
The configurations....

- The algorithms being compared are:
  - HAR
  - CBR, CAP
  - Hybrid schemes: HAR+CBR, HAR+CAP.
  - Baseline (which is without any fancy algorithms)

- The default caching is OPT (optimal restore caching scheme)
- Container size of 4MB
- For HAR:
  - Default utilization threshold of 50%
  - Retain 20 backups in HAR (deleting backup n-20 after backup n)
Speed Factor

- The metric of the restore performance is defined thus:
  - $1/\text{the mean containers read per MB of restored data}$
  - Container size is 4Mb and thus 4 speed factor is the maximum storage bandwidth
    - For every 4MB we read one container, so for every 1 MB we read 0.25 containers, meaning $SF = 1/0.25 = 4$
Tradeoffs

- Deduplication ratio and performance
  - Rewriting harms deduplication ratio
- Deduplication ratio: total data/total data needed to be stored
- Performance: we will focus on restore performance
Tradeoffs

Why would rewriting... harm deduplication ratio

- Deduplication ratio and performance
  - Rewriting harms deduplication ratio
- Deduplication ratio: total data/total data needed to be stored
- Performance: we will focus on restore performance
If you recall....

- Rewriting writes the chunks AGAIN, and points the usage of the chunk to the new write.
- So we SEE that we have a duplicate but WRITE it again thus saving more data and reducing the deduplication ratio.
- We allow multiple copies of the same chunk to be saved.
Datasets in use:

- Two real-world datasets, including VMDK and Linux, and a synthetic dataset, i.e., Synthetic, are used for evaluation.
  - VMDK is from a virtual machine installed Ubuntu 12.04 LTS, which is a common use-case in real-world
- Each dataset is divided into variable-sized chunks.
- In VDMK, OOOC are dominant (a lot of self references)
- In Linux, sparse containers are dominant
- Synthetic streams don’t have self reference

<table>
<thead>
<tr>
<th>dataset name</th>
<th>VMDK</th>
<th>Linux</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>total size</td>
<td>1.44TB</td>
<td>104GB</td>
<td>4.5TB</td>
</tr>
<tr>
<td># of versions</td>
<td>102</td>
<td>258</td>
<td>400</td>
</tr>
<tr>
<td>deduplication ratio</td>
<td>25.44</td>
<td>45.24</td>
<td>37.26</td>
</tr>
<tr>
<td>avg. chunk size</td>
<td>10.33KB</td>
<td>5.29KB</td>
<td>12.44KB</td>
</tr>
<tr>
<td>sparse</td>
<td>medium</td>
<td>severe</td>
<td>severe</td>
</tr>
<tr>
<td>out-of-order</td>
<td>severe</td>
<td>medium</td>
<td>medium</td>
</tr>
</tbody>
</table>
The average utilization of rewriting algorithms.

In VDMK because they rewrite many copies of self-referred chunks, CBR and CAP achieve less than the baseline.
Figure 6: The comparisons between HAR and other rewriting algorithms in terms of deduplication ratio.
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 trigger garbage collection, the actual storage cost is shown later.
- 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%.
- 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.
Severe decline in baseline

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.
Severe decline in baseline

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.

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.
Severe decline in baseline

Opt significantly improves on baseline in VDMK since OOOOC are dominant

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.
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.
CBR, CAP and baseline achieve similar average results in Linux and synthetic datasets since they also have the same average utilizations. (slide 34)

Hybrids decrease cache threshold! And therefore improve performances of smaller caches, which is great! Since that’s why we made them in the first place! They decrease it by X2

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.
CBR, CAP and baseline achieve similar average results in Linux and synthetic datasets since they also have the same average utilizations. (slide 34)

- Hybrids decrease cache threshold! And therefore improve performances of smaller caches, which is great! Since that’s why we made them in the first place! They decrease it by $X^2$.

- In Linux, restore cache is so small that instead of using hybrids, we can use HAR with a reasonable sized restore cache.

Figure 8: The comparisons of rewriting time (in hours) of synthetic and real data. Baseline is the average value of last 20 backups. The cache size is in terms of bytes.
Garbage Collection

- Metadata:
  - We compare the metadata space overhead among existing inline reference management approaches.
  - The metadata overhead of CMA is lowest, and no more than 1/90 of that of GMS!

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

<table>
<thead>
<tr>
<th>Reference Counter [24]</th>
<th>VMDK</th>
<th>Linux</th>
<th>Synthetic</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4.64MB</td>
<td>328.36KB</td>
<td>6.53MB</td>
</tr>
</tbody>
</table>

| GMS [7]                |          |          |           |
|                        | 5.26MB   | 190KB    | 7.23MB    |

| CMA                    | 58.19KB  | 2KB      | 81.62KB   |
We want there to be few of them

HAR achieves better storage saving than the baseline, and the merging is no longer necessary in a deduplication system with HAR.

CAP, CBR increase the problem in VDMK since they keep rewriting the OOOC chunks and they still require merging.
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
- Out-of-order containers 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.
Questions?