Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance

Zhichao Cao, *University of Minnesota;* Shiyong Liu, *Ocean University of China;* Fenggang Wu, *University of Minnesota;* Guohua Wang, *South China University of Technology;* Bingzhe Li and David H.C. Du, *University of Minnesota*

https://www.usenix.org/conference/fast19/presentation/cao

This paper is included in the Proceedings of the
17th USENIX Conference on File and Storage Technologies (FAST '19).

February 25–28, 2019 • Boston, MA, USA

978-1-939133-09-0

Sliding Look-Back Window Assisted Data Chunk Rewriting for Improving Deduplication Restore Performance

Zhichao Cao[†] Shiyong Liu[‡] Fenggang Wu[†] Guohua Wang[§]
Bingzhe Li[†] David H.C. Du[†]

[†]University of Minnesota, Twin Cities [‡]Ocean University of China
[§]South China University of Technology

Abstract

Data deduplication is an effective way of improving storage space utilization. The data generated by deduplication is persistently stored in data chunks or data containers (a container consisting of a few hundreds or thousands of data chunks). The data restore process is rather slow due to data fragmentation and read amplification. To speed up the restore process, data chunk rewrite (a rewrite is to store a duplicate data chunk) schemes have been proposed to effectively improve data chunk locality and reduce the number of container reads for restoring the original data. However, rewrites will decrease the deduplication ratio since more storage space is used to store the duplicate data chunks.

To remedy this, we focus on reducing the data fragmentation and read amplification of container-based deduplication systems. We first propose a flexible container referenced count based rewrite scheme, which can make a better tradeoff between the deduplication ratio and the number of required container reads than that of capping which is an existing rewrite scheme. To further improve the rewrite candidate selection accuracy, we propose a sliding look-back window based design, which can make more accurate rewrite decisions by considering the caching effect, data chunk localities, and data chunk closeness in the current and future windows. According to our evaluation, our proposed approach can always achieve a higher restore performance than that of capping especially when the reduction of deduplication ratio is small.

1 Introduction

With the fast development of new eco-systems such as social media, cloud computing, artificial intelligence (AI), and Internet of Things (IoT), the volume of data created increases exponentially. However, the storage density and capacity increase in main storage devices like disk drives (HDD) and solid-state drives (SSD) cannot match the explosion in the speed of data creation [1, 2]. Data deduplication is an efficient way of improving the storage space utilization such that the cost of storing data can be reduced. Data deduplication is a widely used technique to reduce the amount of data to be transferred or stored in today's computing and communication infrastructure. It has been applied to primary and secondary storage systems, embedded systems, and other systems that host and transfer a huge amount of data.

Data deduplication partitions the *byte stream* formed by the original data into data chunks. The chunk size can be either fixed or variable (e.g., 4KB size in average). For efficiency, deduplicating systems usually process a *segment* of data at one time. A segment is typically the size of several thousand data chunks (e.g., 20MB). Each chunk is represented by a *chunk ID* which is a hashed fingerprint of the chunk content. By representing the original data stream with a sequence of the chunk IDs as well as the metadata to find the data chunks (called a *recipe*) and storing only the unique data chunks (not the duplicates of existing data chunks) into the storage system, the amount of data to be stored can be greatly reduced. The result of deduplication can be measured by the *deduplication ratio* which is the total amount of data in the original byte stream divided by the total size of stored unique data chunks.

Since the size of a data chunk is rather small, storing individual data chunks directly cannot efficiently utilize the storage bandwidth especially for storage systems with low random read/write performance. For HDD based storage systems used by backup or archive applications, the deduplication process typically accumulates a number of data chunks (say 1000 data chunks) in a *container* before writing them out together [3]. For the same reason, when reading a data chunk to restore the original data, the whole container is read from storage since we are expecting that many data chunks in the same container will be accessed soon. In this paper, we focus on the container-based deduplication systems.

The data deduplication restore process accesses and returns the data chunks based on the order in the recipe to recreate the original byte stream. Since the unique data chunks are writing to storage based on the order of their first appearance in the byte stream, a duplicate data chunk may require reading a container that was stored a long time ago. In this case, the restore process may not require most of the data chunks in such a container in the near future. This causes data fragmentation (i.e., data chunks are scattered) and read amplification (i.e., the size of data being read is larger than

the size of data being restored), which leads to low restore performance. Therefore, reducing the number of container reads is a major task for restore performance improvement.

To improve the restore performance, several techniques have been proposed including caching schemes (e.g., container-based caching [4, 5, 6, 7], chunk-based caching [8, 9, 10], and forward assembly [8, 10]) and duplicate data chunk rewrite schemes [4, 5, 6, 7, 8]. Since the unique data chunks are stored in the same order as they first appeared, unique data chunks in the same container are usually at nearby locations that these chunks are first identified in the original byte stream. If a large number of data chunks from the same container are used in a small range in the original byte stream, caching schemes can be very useful in reducing the number of container reads.

When a deduplication system runs for a long time, the data chunks in a segment of the original byte stream can be highly fragmented. That is, the duplicate data chunks in this segment are stored in a large number of containers that are new or have already existed (old containers). When restoring this segment, each container read from storage (read-in container) holds only a few data chunks that belong to this segment. Therefore, a large number of data chunks are read in from storage, possibly evicting cached data chunks before they are used again in the restore process. In this scenario, the number of container reads cannot be effectively reduced by applying caching schemes only.

In this situation, each container read only contributes a small number of data chunks for restore. This type of container read is inefficient and expensive. To address this issue, rewriting some of the duplicate data chunks in a highly fragmented segment and storing them together with the nearby newly identified unique data chunks in the same container can effectively reduce the number of container reads required for restoring these chunks. However, it is hard to decide which duplicate data chunks to rewrite during the deduplication process due to the limited information that can be used by a rewrite scheme.

Previous studies introduced several data chunk rewriting policies. Among these studies, Lillibridge *et al.* [8] proposed a simple but effective data chunk rewriting selection policy, named *capping*. Capping first partitions the byte stream into fixed size segments (e.g., 20MB). Then, in each segment, the referenced old containers are sorted in a descending order based on the number of data chunks in each container that appeared in the segment. Data chunks in the old containers which are ranked out of a pre-defined threshold (i.e., *capping level*) are stored again (to be *rewritten*). Therefore, when restoring the data chunks in this segment, the total number of container reads is limited by the capping level plus the number of new containers created when deduplicating this segment.

Capping operates on a single fixed-size segment of input data at a time. It applies the same cap to each segment and analyzes each segment in isolation, without considering the contents of the previous or following segments. Also, the deduplication ratio cannot be guaranteed via capping. In this paper, we explore two data chunk rewrite schemes to improve restore performance. First, based on capping, we propose a flexible container referenced count based scheme to adjust the cap for each segment according to the fragmentation of the duplicate data chunks in that segment. Second, we propose a new rewrite scheme called Sliding Look-Back Window Rewrite that makes better rewrite decisions by considering a larger amount of input data around each duplicate chunk. This avoids inefficiencies when related duplicate chunks are divided by the arbitrary boundaries between segments. The new rewrite policy for the sliding look-back window scheme combines the flexible container referenced count based scheme with the consideration of caching effects in the restore process.

To fairly and comprehensively evaluate our rewriting designs, we implemented a system with both deduplication and restore engines for normal deduplication, the capping scheme, the two schemes we are proposing, Forward Assembly (FAA) [8], and a chunk-based caching scheme called ALACC [10]. We compared and evaluated the performance of different combinations of deduplication and restore engines. We use *speed factor* (MB/container-read), which is defined as the mean size data being restored (MB) per container read [8], to indicate the amount of data that can be restored by one container read on average. Speed factor is platform independent and a higher speed factor usually indicates a higher restore performance. Using several real-world deduplication traces, with the same deduplication ratio and the same restore engine, our proposed sliding look-back window based design always achieves the best restore performance. Our design can improve the speed factor up to 97% compared with normal deduplication and it can improve the speed factor up to 41% compared with capping.

The rest of the paper is presented as follows. Section 2 reviews the background of data deduplication and the related work of caching and rewrite schemes. Section 3 describes a flexible container referenced count based rewrite scheme which improves on capping. To further reduce the number of container reads, a sliding look-back window based rewrite scheme is proposed and presented in Section 4. Based on the sliding look-back window, the rewrite candidate selecting policy is discussed in Section 5. We present the evaluation results and analysis in Section 6. Finally, we conclude our work and discuss future work in Section 7.

2 Background and Related Work

In this section, we first briefly describe the deduplication and restore processes. Then, the related studies of improving the restore performance are presented and discussed.

2.1 Deduplication and Restore Process

Data deduplication is widely used in secondary storage systems, such as archiving and backup systems, to improve storage space utilization [3, 11, 12, 13, 14, 15, 16, 17]. Data deduplication is also deployed in primary storage systems to make better trade-offs between cost and performance [18, 19, 20, 21, 22, 23, 24, 25]. After the original data is deduplicated, only the unique data chunks and the recipe are stored. When the original data is requested, the recipe is used to read the corresponding data chunks for assembling the data. From the beginning of the recipe, the restore engine uses the data chunk metadata to access the corresponding data chunks one by one and assembles the data chunks in the memory buffer (assembling buffer). Once the engine accumulates a buffer worth of data chunks, it returns the restored data.

To store the unique data chunks, some deduplication systems such as HYDRAstor [17], iDedup [18], Dmdedup [22], and ZFS [26] directly store individual data chunks to the persistent storage. They do not incur read amplification during restore, but they suffer from data chunk fragmentation and the slow performance of random reads. Other deduplication systems, such as the backup products from Veritas [27] and Data Domain [3], pack a number of data chunks (compression may be applied) in one I/O unit (called a container in this paper). Data chunks in the same container are written out and read in together to benefit from the high sequential I/O performance and good chunk localities. In this paper, we focus on the container-based deduplication systems.

In the worst case of restore, we may need N container reads to assemble N data chunks. A straightforward way to reduce the number of container reads is to cache some of the containers or data chunks. Since some data chunks will be used soon after they are read into memory, these cached chunks can be directly copied from the cache to the assembling buffer which can reduce the number of container reads. In some scenarios, even with caching schemes like chunk-based caching and forward assembly [8], the number of container reads cannot be further reduced. This is especially true for restoring a duplicate data chunk since this data chunk was stored in a container created earlier and most of the data chunks in this container may not be needed by the restore process.

Another way to reduce the number of container reads is to store (rewrite) some of the duplicate data chunks together with the unique chunks in the same container during the deduplication process. The decision to rewrite a duplicate data chunk has to be made during the deduplication process instead of being done at restore process like caching. After rewriting, the duplicate chunks and unique chunks will be read together from the same container, thus avoiding the need to read these duplicate chunks from other old containers. However, this approach reduces the effectiveness of data deduplication (i.e., reduces the deduplication ratio). Rewrite can effectively reduce the number of container reads in the cases that caching schemes do not work well, especially when each container read only contributes a few data chunks to restoring the original data.

2.2 Related Work on Restore Performance Improvement

We will first review the different caching policies such as container-based caching, chunk-based caching, and forward assembly. Then, we will focus on the studies of storing duplicate chunks to improve restore performance.

Different caching policies are studied in [4, 5, 6, 8, 9, 10, 28]. To improve the cache hit ratio, Kaczmarczyk *et al.* [4], Nam *et al.* [5, 6], and Park *et al.* [28] used container-based caching, which cache containers in memory. Container-based caching schemes can achieve Belady's optimal replacement policy [7]. To achieve a higher cache hit ratio, some studies cache data chunks directly [8, 9]. If we compare container-based with chunk-based caching, the former has lower cache management overhead (e.g., fewer memory copies), while the latter has a higher cache hit ratio. Therefore, caching chunks is preferred in the restore process, especially when the cache space is limited. Lillibridge *et al.* proposed the forward assembly scheme, which reads ahead in the recipe and pre-fetches some chunks into a Forward Assembly Area (FAA) [8]. This scheme ensures that no container will be read more than once when restoring the data chunks of the current FAA. The management overhead of FAA is lower than that of chunk-based caching, but the restore performance of these two schemes is closely related to the workload characteristics. Therefore, we previously proposed a new caching scheme which combines FAA and chunk-based caching called Adaptive Look-Ahead Chunk Caching (ALACC). ALACC can potentially adapt to various workloads and achieve better restore performance [10].

Nam *et al.* [5, 6] first introduced the Chunk Fragmentation Level (CFL) to estimate degraded read performance of deduplication storage. CFL is defined as a ratio of the optimal number of containers with respect to the actual number of containers required to store all the unique and duplicate chunks of a backup data stream. When the current CFL becomes worse than a predefined threshold, data chunks will be rewritten. Kaczmarczyk *et al.* [4, 29] utilized stream context and disk context of a duplicate block to rewrite highly-fragmented duplicates. A data chunk whose stream context in the current backup is significantly different from its disk context will be rewritten. Fu *et al.* [7, 30] proposed a History-Aware Rewriting algorithm (HAR) which identifies and rewrites sparse containers according to the historical information of the previous backup. A new rewrite scheme called container capping was proposed by Lillibridge *et al.* [8], which uses a fixed-size segment to identify the data chunks to be rewritten. Since each container read involves a

large fixed number of data chunks, Tan *et al.* [31] proposed a Fine-Grained defragmentation approach (FGDefrag) that uses variable-size data groups to more accurately identify and effectively remove fragmented data. FGDefrag rewrites the fragmental chunks and the new unique chunks of each segment into a single group.

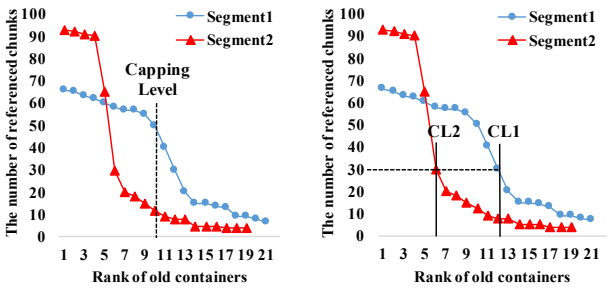
In these data chunk rewrite schemes, after some of the data chunks are rewritten, one data chunk can appear in several different containers. How to choose one container to be referenced is challenging. Wu *et al.* [32, 33] proposed a cost-efficient rewriting scheme (SMR). SMR first formulates the defragmentation as an optimization problem of selecting suitable containers and then builds a sub-modular maximization model to address this problem by selecting containers with more distinct referenced data chunks.

Caching schemes can be effective if the data chunks of the read-in container will be used again in the near future during the restore. If duplicate data chunks are spread across many containers, a large number of container reads will be required. Caching schemes cannot reduce these compulsory misses and reads. Thus, rewriting some of the duplicate data chunks that can effectively reduce the number of container reads is an alternative solution. However, it is difficult to make decisions on which duplicate chunks need to be rewritten with the minimum reduction of the deduplication ratio. This is the focus of our study.

As mentioned before, container capping is a simple and effective rewrite scheme [8]. In this scheme, the original data stream is partitioned into fixed size segments, and each segment has the size of N containers (e.g., $N \cdot 4MB$). The goal of this scheme is to limit the number of container reads required to restore a segment of data to $T+C$. C is the number of new containers generated when deduplicating the segment, and T is the *capping level*. Thus, the capping level limits the number of containers that will be read to load duplicate chunks. In capping, the first step is to count the number of data chunks (including duplicates) in the segment that belongs to an old container, called *CoNtainer Referenced Count* (CNRC). Then, these old containers containing at least one duplicate data chunk in the segment are sorted with their CNRCs in descending order. If the container is ranked lower than T, the duplicate data chunks in this container are written together with unique data chunks into the currently active container. In this way, capping ensures the higher bound of container reads in each segment. However, capping also has limitations, which motivate us to design new rewrite schemes for higher restore performance. The details are presented in the following.

3 Flexible Container Referenced Count based Design

In this section, we first discuss the limitations of the capping scheme. Then, we propose a flexible container referenced count based scheme which improves the performance of capping and achieves a good tradeoff between the number of container reads and the deduplication ratio.



(a) The CNRCs distribution and the fixed capping level

(b) The CNRCs distribution and the specified capping level for different segments

Figure 1: An example of the relationship between the CNRCs distribution and the capping level

3.1 Limitations of Capping

Capping uses a fixed threshold to control the capping level for all the segments such that a higher bound on the number of container reads can be obtained. However, the deduplication ratio is not considered as an optimization objective in the capping scheme. We have found that with some changes to the capping scheme, a better tradeoff between the number of container reads and the data deduplication ratio can be achieved.

If we consider the number of old containers involved in a segment and the distribution of CNRCs of these old containers, we find that the number of old containers and the distribution of CNRCs can be very different for different segments. If we wish to bound the reduction of deduplication ratio, some segments can keep fewer container reads than the targeted capping level while other segments may have to go beyond the targeted capping level. For example, in a segment, after sorting the old containers according to their CNRCs in descending order, we can plot out the CNRC distribution of these containers as shown in Figure 1(a). The X-axis is the rank of containers and Y-axis is the number of referenced data chunks of an involved container. Different segments have different distributions. Consider Segment 1 and Segment 2 in Figure 1(a). A fixed capping level of 10 is shown. All duplicate data chunks in the containers to the right of the capping level have to be rewritten.

In this example, the number of container reads of these two segments is capped by 20 plus the number of newly generated containers. For Segment 1, containers ranked from 10 to 12 have a relatively large number of referenced data chunks. Rewriting duplicates of data chunks in these containers will cause more reduction of deduplication ratio. The ideal capping level for Segment 1 should be either 12 or 13 according to the distribution. For Segment 2, since data

chunks are more concentrated in the higher ranking containers, rewriting the duplicate data chunks in the containers ranked beyond container 6 or 7 instead of container 10 will reduce more container reads while the increased amount of duplicate data chunks being rewritten is limited. Therefore, as shown in Figure 1(b), if we use 12 as the new capping level for Segment 1 (CL1 in the figure) and 6 as the new capping level for Segment 2 (CL2 in the figure), the number of container reads will be 1 fewer and the number of duplicate data chunks being rewritten will be even lower. Therefore, applying varied capping levels for different segments can further reduce container reads and achieve even fewer data chunk rewrites. However, how to decide the "capping levels" for different segments is challenging.

3.2 Flexible Container Referenced Count Scheme

To address the aforementioned limitations of using a fixed capping level, we proposed the Flexible Container Referenced Count based scheme (FCRC). It is an improvement of capping. Instead of using a fixed capping level as the selection threshold, FCRC uses a value of CNRC as the new threshold. It rewrites duplicate data chunks from old containers that have CNRCs lower than the threshold. In this way, different segments will have different actual capping levels. The actual capping levels are decided by the threshold and the distribution of CNRCs of these segments. The CNRC threshold T_{cnrc} can be estimated by a targeted capping level. That is, the total number of duplicate data chunks (including all copies of a duplicate) in a segment divided by the targeted capping level. Statistically, if each read-in container in a segment can contribute more than T_{cnrc} number of duplicate data chunks, the total number of old container reads to restore this segment will be bounded by the targeted capping level. Thus, rewriting duplicate data chunks in a container with CNRC lower than T_{cnrc} can achieve a similar number of container reads as using the same targeted capping level in the capping scheme.

Using a fixed value of T_{cnrc} as a threshold can make a tradeoff between the number of container reads and the deduplication ratio, but it cannot guarantee either an upper bound on the number of container reads, like capping, or a lower bound on the deduplication ratio. T_{cnrc} can only decide which data chunks belonging to the old containers with low CNRC are rewritten. The actual number of duplicate data chunks being rewritten in each segment is unknown. Also, the estimation of T_{cnrc} does not guarantee the minimal or maximal number of container reads. To solve this issue, we use a targeted deduplication ratio reduction limit $x\%$ and the targeted number of container reads Cap in one segment to generate these two bounds for T_{cnrc}. In the following, we will discuss the algorithm to calculate the two bounds and how we decide the T_{cnrc} of one segment. The old containers referenced in one segment are sorted by CNRCs in descending order before we start the following algorithm.

Bound for Deduplication Ratio Reduction We first calculate the targeted number of data chunks that can be rewritten in total (N_{rw_total}) according to the deduplication ratio reduction limit $x\%$ (i.e., after rewrite, deduplication ratio is at most x% lower than that of no the rewrite case). Let us consider a backup system as an example. Suppose in the current backup version, the total number of data chunks is N_{dc} and the deduplication ratio reduction limit is x%. For one backup version, we use the number of unique data chunks generated in the previous deduplicated version as an estimate of that value for the current version, which is N_{unique}. If we do not rewrite duplicate data chunks, the deduplication ratio is $DR = \frac{N_{dc}}{N_{unique}}$. To have at most x% of deduplication ratio reduction, we need to rewrite at most N_{rw_total} data chunks in total after deduplication. We can calculate the new deduplication ratio after rewrites by $DR \cdot (1 - x\%) = \frac{N_{dc}}{N_{unique} + N_{rw\_total}}$. Finally, we have $N_{rw\_total} = \frac{N_{unique} \cdot x\%}{1 - x\%}$. By dividing N_{rw_total} by the total number of segments in this backup version, we can calculate the average data chunks that can be rewritten in each segment, which is H_{rw}. H_{rw} is used for all the segments in one backup. For other applications, we can run deduplication for a short period of time. Then, we estimate H_{rw} using the current total number of data chunks being generated as N_{dc} and the current total number of unique chunks as N_{unique}.

Since the actual rewrite number can be smaller than H_{rw} in some segments, we can accumulate and distribute the saving as credits to the rest of the segments such that some of the segments can rewrite more than H_{rw} duplicate data chunks. In this way, the number of container reads can potentially be reduced, but the overall deduplication reduction limit x% is still satisfied. Note that, we cannot allow rewriting more data chunks than planned and hope segments in the future can pay for the deficit. So the accumulated credit is always non-negative.

For the i_{th} segment, suppose the accumulated actual rewrites before i_{th} segment is N_{rw}^{i-1}. We can rewrite at most $H_{rw} \cdot i - N_{rw}^{i-1}$ duplicate data chunks in the i_{th} segment. If $N_{rw}^{i-1} = H_{rw} \cdot (i-1)$ (the accumulated credit is 0), we still use H_{rw} as the rewrite limit for the i_{th} segment. In this way, we get the referenced count bound RC_{rw}^{i} of the i_{th} segment by adding the CNRCs of the old containers from low to high until the sum reaches the rewrite limit.

Bound for Container Reads Suppose the bound on the number of container reads of the i_{th} segment is RC_{reads}^{i}. The maximum number of old containers being referenced in one segment is Cap, which is the same concept as capping level in capping scheme. Suppose the accumulated number of old containers referenced before the i_{th} segment is N_{reads}^{i-1}. For the i_{th} Segment, we can tolerate referencing at most $Cap \cdot i - N_{reads}^{i-1}$ containers. Counting the old containers ranking from high to low, when the container number is $Cap \cdot i - N_{reads}^{i-1}$, the CNRC of this container is RC_{reads}^{i}. It is possible that $Cap \cdot i - N_{reads}^{i-1}$ is higher than the number of referenced old

containers in this segment. In this case, $RC^i_{reads} = 0$ and the credit is accumulated for the future.

T_{cnrc} **Calculation** If $RC^i_{rw} < RC^i_{reads}$ in the i_{th} segment, we are unable to satisfy the number of container reads and the targeted deduplication reduction limit at the same time. We choose to satisfy the deduplication ratio reduction limit first. Therefore, the threshold T_{cnrc} that we use in this segment is RC^i_{rw}. The bound on the number of container reads will be violated and the credits of the number of container reads of this segment will be negative. If $RC^i_{rw} \geq RC^i_{reads}$, the threshold T_{cnrc} can be adjusted between RC^i_{rw} and RC^i_{reads}. If T_{cnrc} of the previous segment is in between, we use the same T_{cnrc}. Otherwise, we use $\frac{RC^i_{rw}+RC^i_{reads}}{2}$ as the new T_{cnrc} which can accumulate the credit for both the number of data chunks rewritten and the number of container reads.

The performance comparisons between the FCRC scheme and the capping scheme are shown in Section 6. In general, by using a flexible referenced count based scheme, the container reads are effectively reduced compared to the capping scheme with a fixed value of capping level when the same deduplication reduction ratio is achieved.

4 Sliding Look-Back Window

Although our proposed FCRC scheme can address the trade-off between the number of container reads and the deduplication ratio, using a fixed size segment partition causes another problem. In this section, we will first discuss the problem. Then we present a new rewrite framework called Sliding Look-Back Window (LBW) in detail.

4.1 Issues of Fixed Size Segment Partition

In both the capping and FCRC schemes, the decision to rewrite duplicate data chunks near the segment partition boundaries may have issues. Let us look at the example shown in Figure 2. There are two consecutive segments $S1$ and $S2$. The majority of data chunks of container $C1$ appear at the front of $S1$ and the majority of data chunks of $C2$ are at the front of $S2$. Due to the segment partition point, a few duplicate chunks of container $C1$ are also at the front of $S2$ and a few duplicate chunks of container $C2$ are at the end of $S1$. According to the capping scheme, the number of data chunks of $C1$ in $S1$ and that of $C2$ in $S2$ are ranked higher than the capping level. These chunks will not be rewritten. However, the ranking of container $C1$ in $S2$ and container $C2$ in $S1$ are out of the capping level. Since we do not know the past information about $C1$ in $S1$ when deduplicating $S2$ and the future information about $C2$ in $S2$ when deduplicating $S1$, these chunks (i.e., a few data chunks from $C2$ in $S1$ and a few data chunks from $C1$ in $S2$) will be rewritten. When we restore $S1$, $C1$ and $C2$ will be read in the cache. When restoring $S2$, $C2$ is already in the cache and the additional container read will not be triggered. Therefore, rewriting the

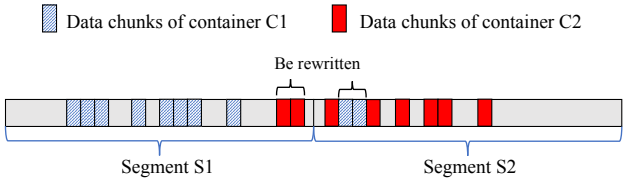


Figure 2: An example to show the fixed size segment partition issue in capping

data chunks of $C2$ appearing in $S1$ and a few data chunks of $C1$ in $S2$ is wasted.

As demonstrated in the aforementioned example, in both the capping and FCRC schemes, the container and data chunk information based on a fixed size segment do not include information about the past and future segments. On one hand, data chunks close to the front of a segment are evaluated together with subsequent chunks in the same segment, but they do not benefit from information about data chunk references and rewrite decisions made in the previous segment. On the other hand, data chunks close to the end of the segment are evaluated with earlier chunks in the segment, but there is no information about the next segment that can be used. The data chunks close to the end of the segment have a higher probability of being rewritten if most of the data chunks of their containers appear in the subsequent few segments. As a result, the rewrite decisions made with the statistics only from the current segment are less accurate.

4.2 Sliding Look-Back Window Assisted Rewrite

To prevent various cases at the boundaries of the segments and to more precisely evaluate each data chunk in a segment, we propose a fixed size "sliding" window design. In the deduplication process, the window covers a range of data chunks that have been recently generated by the deduplication engine and slides forward to cover the newly generated data chunks. A newly generated data chunk will start at the front of the sliding window. At this moment, each data chunk can be evaluated with a window-size of "past information" to make an initial rewrite decision, or a rewrite decision for this data chunk can be made later before it moves out the window. As the deduplication process continues, the window moves forward. Before a previously generated data chunk moves out of the sliding window, we can evaluate it with a window-size of "future information" to make the final rewrite decision. In this design, all data chunks are fairly evaluated with one window-size of past information and one window-size of future information which solves the issues of fixed segment partition. The details of the proposed sliding look-back window assisted rewrite scheme are presented in the following.

The overall architecture is shown in Figure 3. To move the window efficiently, the sliding window consists of N containers (4 containers of three chunks each in this example)

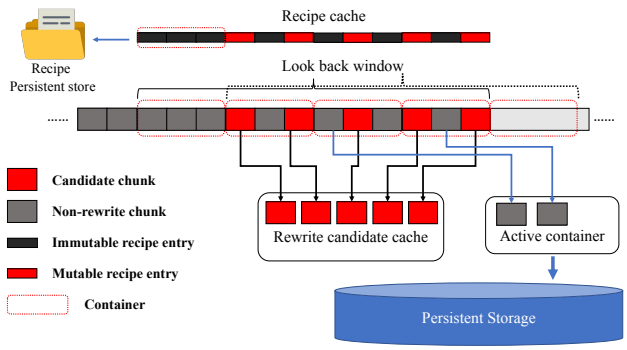


Figure 3: The overview of the sliding look-back window assisted rewrite design

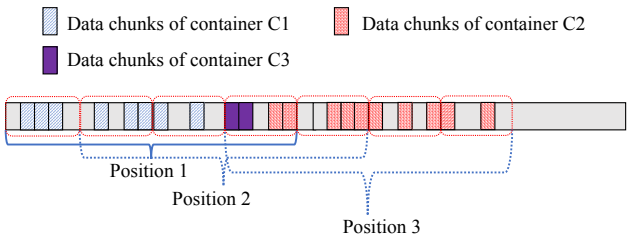


Figure 4: An example of the sliding look-window assisted rewrite scheme

and it is moved forward in container granularity. When one container size new data chunks are accumulated, they are added to the front of the window together. At the same time, one container size data chunks at the end of the window are moved out. The window moves forward to process the newly generated data chunks while these data chunks are evaluated together with other data chunks in the back of the window to make the initial rewrite decisions. This is why we call it a *Look-Back Window* (LBW). The LBW acts as a recipe cache that maintains the metadata entries of data chunks in the order covered by the LBW in the byte stream. The metadata entry for each data chunk consists of the same information found in a recipe referencing the data chunk: chunk metadata, offset in the byte stream, container ID/address, and the offset in the container. This information is used to select the data chunks to be rewritten.

To implement such a look back mechanism, some data needs to be temporarily cached in memory. In our design, we cache three types of data in memory as shown in Figure 3. First, as mentioned before, the LBW maintains a recipe cache. Second, similar to the most container based deduplication engines, we maintain an active container as chunk buffer in memory to store the unique and rewritten data chunks. The buffer is of one container size. Third, we maintain a rewrite candidate cache in memory to temporally cache the candidate data chunks. The size of the rewrite candidate cache is user-configurable. The same chunk may appear in different positions in the window, but only one copy is cached in the rewrite candidate cache. As the window moves, data chunks in the rewrite candidate cache will be gradually evicted according to the rewrite selection policy and these chunks become non-rewrite chunks. Finally, before the last container is moved out, the remaining candidate chunks in that container are rewritten to the active container. The metadata entry of a candidate chunk is mutable while the metadata entry of a non-rewrite chunk is immutable. The details of the rewrite selection policy will be presented in Section 5.

We perform the following steps to move the LBW forward. First, as each new data chunk is received, it is identified as unique or duplicate and added to the first container of the LBW. If a data chunk is unique, its data will be put into the active container and its corresponding metadata entry is stored in the recipe cache. If the data chunk is a duplicate, its metadata entry is added to the recipe cache and temporarily references the old container. Then, we make an initial rewrite decision for this data chunk. If a data chunk is a duplicate satisfying the non-rewrite condition (described later), it will not be rewritten and will be marked as a *non-rewrite chunk*. In this case, the data chunk will not be added to the rewrite candidate cache. If the rewrite decision cannot be made at this moment, this data chunk will be added to the rewrite candidate cache as one *candidate chunk* and we will wait for more information. When the most recent container in the LBW fills up with data chunks, the oldest container in the LBW (at the end of the LBW) is moved out and its corresponding metadata entries are written to the recipe on storage. Before the oldest container is moved out, the rewrite decisions of all candidate chunks in that container have to be made. Then, the LBW will move forward to cover a new container size of data chunks.

The effect of delaying rewrite decisions using a look-back window is as follows. When a data chunk is identified as a candidate chunk, we will have the information of all data chunks in the past $N-1$ containers including old container referencing information and the data chunk rewrite decisions of these data chunks. If we cannot make a decision for this candidate chunk immediately due to the limited information, we can wait to make a decision before this chunk is moved out of the window. At that time, we will know the data chunks subsequent to this data chunk in the future $N-1$ containers. With both past and future information in the LBW, a more accurate decision can be made.

Let us consider the simple example as shown in Figure 4. Suppose we use a fixed reference count value of 4 as the threshold. That is, once an old container has more than 4 duplicate data chunks in the window, the duplicate data chunks in that container will not be rewritten (non-rewrite condition). When the LBW is at Position 1, the number of data chunks from Containers 1, 2, and 3 are 8, 2, and 2, re-

spectively. Based on the non-rewrite condition, the chunks from Container 1 will not be rewritten. The newly identified data chunks from Containers 2 and 3 cannot satisfy the non-rewrite condition. We add these data chunks to the rewrite candidate cache. When the LBW is moved to Position 2, the number of data chunks from Container 2 is already 5, which satisfy the non-rewrite condition. The data chunks from Container 2 will not be rewritten. Therefore, the data chunks of Container 2 in the rewrite candidate cache are dropped. For the data chunks from Container 3, we still need to delay to make the rewrite decision. When the LBW is moved to Position 3, the data chunks from Container 3 are already at the end of the LBW. At this moment, the number of data chunks from Container 3 is still lower than the threshold. Thus, the two data chunks from Container 3 are rewritten to the active container. At the same time, the corresponding metadata entries in the recipe cache are updated with the active container information. As the oldest container is moved out of the LBW, the corresponding metadata entries are written to storage.

5 Rewrite Selection Policy for LBW

In the LBW scheme, since there are no segment partitions, we cannot directly sort the old containers by their CNRCs, which vary as the LBW moves. Therefore, either the capping level or the threshold T_{cnrc} of FCRC cannot be directly used in LBW. Based on the flexible container referenced count concept proposed in FCRC, we design a new rewrite selection policy for LBW. In the new policy, we further consider the cache-effective range and the container read efficiency of the restore process. These two criteria help us to adjust the threshold and make more accurate rewrite decisions. In this section, we first discuss the two criteria and how we can use them during the deduplication process to help make the rewrite decisions of duplicate data chunks. Then, the details of the rewrite policy are described.

5.1 Two Criteria of Restore Process

In most data chunk rewrite studies, caching-effectiveness of the restore is not considered. Some data chunks being rewritten may actually be in the cache when these chunks are restored, causing unnecessary rewrite overhead. On the other hand, how many data chunks can be restored by one container read and how long these chunks will stay in the cache can also influence the rewrite decision. Therefore, we bridge the deduplication and restore processes with the two criteria and design a new rewrite policy for LBW.

Cache-Effective Range In most data deduplication systems, caching is used to improve the restore performance. Different caching policies (e.g., chunk-based caching, container-based caching, or FAA) have different eviction priorities. However, for a very short period of time, the data chunks of a read-in container will not be evicted. In other words, if a container is read into the memory, the data chunks of that container will be cached for at least the next S container size data chunks of restore, we consider this the cache-effective range. After restoring S container size data chunks, these cached data chunks will be gradually evicted. For FAA, S is the size of FAA in terms of the number of containers. For chunk-based caching and container-based caching, S is closely related to the size of the cache and varies as the workload changes. Although it may be hard to precisely know when a data chunk will be evicted, we can still estimate S as a lower bound on the cache-effective range of each read-in container.

Importantly, the size of the LBW is related to the cache-effective range S. On one hand, if the size of the LBW is much smaller than S, some rewritten data chunks may have a probability to be in the cache. This will cause an unnecessary reduction of the deduplication ratio. On the other hand, if LBW is much larger than S, a data chunk that was not rewritten due to the same data chunk existing in LBW may not be cached during the restore process (i.e., should have been rewritten). Also, a large LBW requires maintaining a larger amount of metadata in memory and caching a larger number of candidate chunks, which may not be acceptable. Therefore, maintaining an LBW with the size compatible with the cache-effective range is a good tradeoff.

Container Read Efficiency A container's read efficiency can be measured by the number of data chunks used in a short duration after it is read-in and the concentration level of these chunks in a given restore range. Please note that at different moments of the restore, the same old container may have a different read efficiency. If one container read can restore more data chunks, that container read is more efficient than rewriting those data chunks. With the same number of data chunks being restored from one old container, if those data chunks are more concentrated in a small range of the byte stream, fewer data chunks will be cached and the cached chunks can be evicted earlier. In this case, the cache space can be more efficiently used and more container reads can be potentially eliminated.

To quantitatively define a container read efficiency, we use two measurements of an old container in the LBW: *container referenced count* and *referenced chunk closeness*. The container referenced count (CNRC) was used in FCRC to decide the rewrite candidates in a segment. Similarly, we define the container referenced count of an old container as the number of times the data chunks in that container appear in the LBW (duplications are counted) at a point in time, which is $CNRC_{lbw}$. $CNRC_{lbw}$ of an old container can change with each movement of the LBW as data chunks are added in and moved out. In fact, the nature of capping is to rank the $CNRC_{lbw}$ of the old containers appearing in a segment and to rewrite the data chunks that belong to the old containers with low CNRC. The FCRC scheme rewrites the containers with CNRC lower than the CNRC threshold.

We define the referenced chunk closeness L_c as: the average distance (measured by the number of data chunks) between a data chunk that will potentially trigger a container read and the rest of the data chunks of that container in the same LBW, divided by the size of LBW (the total number of chunks in the LBW), where $L_c < 1$. Note that if one data chunk appears in multiple positions, only the first one is counted. Smaller L_c means the data chunks are closer.

5.2 Rewrite Selection Policy for LBW

The rewrite selection policy is designed based on the information covered by the LBW. Suppose the LBW size is S_{LBW} which is measured by the number of containers. When the LBW moves forward for one container size, a container size of data chunks (added container) will be added to the front of the LBW and one container size of data chunks (evicted container) will be removed from the end of the LBW. There are five steps to make the rewrite decision: 1) process the added container, classify the data chunks into three categories: unique chunks, non-rewrite chunks (duplicate data chunks that will not be rewritten), and candidate chunks (duplicate data chunks that may be rewritten); 2) update the metadata entries of data chunks in the added container and add the identified candidate chunks to the rewrite candidate cache; 3) recalculate the $CNRC_{lbw}$ of old containers that contain the data chunks in the rewrite candidate cache and reclassify these data chunks according to the updated $CNRC_{lbw}$ and the threshold T_{dc}. At this step, some of the data chunks in the candidate cache are identified as non-rewrite chunks and are evicted from the cache; 4) rewrite the remaining candidate chunks in the evicted container to the active container and update their metadata entries in LBW. The updated metadata entries of data chunks in the evicted container are written to the recipe persistently; 5) every S_{LBW} containers movement of the LBW, adjust the CNRC threshold T_{dc}, which is used to filter out the non-rewrite chunks. The details of each step are explained in the following.

In **Step 1**, by searching the indexing table, the unique data chunks and duplicate data chunks are identified. The unique data chunks are written to the active container. Next, for each duplicate data chunk, we search backward in the LBW. If some data chunks from the same old container appear ahead of that duplicate data chunk and these chunks are non-rewrite chunks, the duplicate data chunk is marked as a non-rewrite chunk. Otherwise, the duplicate chunk becomes a candidate chunk and it is added to the candidate cache. If a candidate chunk is the first chunk of its container appearing in the most current S containers, it means this container has not been used or referenced at least S containers range. This chunk will potentially trigger one container read and thus we call it the *leading chunk* of this container. As the LBW moves, we can determine the range and distance of data chunks from the same container appearing after the leading chunk. Based on future information, we can decide whether these data chunks should be rewritten or not.

In **Step 2**, the metadata entries of data chunks in the added container are updated with the metadata information of the referenced chunks in the active container or old container. Note that the same duplicate data chunk can appear in multiple old containers due to past rewrites. One of the old containers should be referenced in the metadata entry. There are some studies on how to optimize the selection of old containers. For an example, an approximately optimal solution is proposed by [32, 33]. Here, when the indexing table returns the list of old containers that store a data chunk, we use a greedy algorithm to select the container that has the most chunks in the current LBW. Other algorithms can also be easily applied to our proposed policy.

In **Step 3**, we follow the definition of $CNRC_{lbw}$ described in Section 5.1 to calculate the latest $CNRC_{lbw}$ of each old container referenced by the candidate chunks. The candidate chunks from the old containers whose $CNRC_{lbw}$ are higher than the threshold T_{dc} (the calculation of T_{dc} will be discussed in Step 5) are removed from the rewrite candidate cache. They become non-rewrite chunks and their metadata entries in the recipe cache still reference the old containers.

In **Step 4**, the data chunks in the evicted container that are still in the candidate cache are rewritten to the active container before we move the evicted container out of the LBW. Their metadata entries in the recipe cache are updated to reference the active container. If a leading chunk of one old container is rewritten, which means the $CNRC_{lbw}$ of its container is still lower than T_{dc}, other data chunks from the same container appearing in the current LBW will be rewritten too. Once a data chunk is rewritten, it is evicted from the candidate cache.

In **Step 5**, T_{dc} is adjusted to make better tradeoffs between deduplication ratio reduction and the number of container reads. Every S_{lbw} size movement is one LBW moving cycle and the T_{dc} is adjusted at the end of each cycle. Similar to the FCRC scheme in Section 3.2, we calculate the two bounds of T_{dc} in the i_{th} moving cycle: RC^i_{rw} and RC^i_{reads}. The referenced chunk closeness of current and previous moving cycle are L^i_c and L^{i-1}_c and they are used to further adjust the threshold according to the data chunk closeness. The algorithm details to calculate T_{dc} are described in **Algorithm 1**. Finally, we get the threshold T^{i+1}_{dc} for the $i+1_{th}$ moving cycle.

6 Performance Evaluation

To comprehensively evaluate our design, we implemented four deduplication engines including normal deduplication with no rewrite (Normal), capping scheme (Capping), flexible container referenced count based scheme (FCRC), and Sliding Look-Back Window scheme (LBW). For the restore engine, we implemented two state-of-the-art caching designs, Forward Assembly Area (FAA) [8] and Adaptive

Algorithm 1 T_{dc} Adjusting Algorithm

Input: $T^i_{dc}, L^i_c, L^{i-1}_c, RC^i_{rw}, RC^i_{reads}$
Output: T^{i+1}_{dc}
  if $RC^i_{rw} < RC^i_{reads}$ **then**
    $T^{i+1}_{dc} \leftarrow RC^i_{rw}$
  else
    if $RC^i_{reads} < T^i_{dc}$ **AND** $T^i_{dc} < RC^i_{rw}$ **then**
      $T^i_{start} \leftarrow T^i_{dc}$
    else
      $T^i_{start} \leftarrow (RC^i_{reads} + RC^i_{rw})/2$
    end if
    if $L^i_c < L^{i-1}_c$ **then**
      $T^{i+1}_{dc} \leftarrow T^i_{start} - 1$
    else
      $T^{i+1}_{dc} \leftarrow T^i_{start} + 1$
    end if
  end if

Look-Ahead window Chunk based Caching (ALACC) [10]. Since any restore experiments will include one deduplication engine and one restore engine, there are 8 combinations. Speed factor and deduplication ratio are used as the evaluation metrics, which are determined by the workload, deduplication and restore engine designs. The two metrics are platform independent. Since the container I/O time dominates the whole restore time, especially when low performance storage (e.g., HDD or tape) are used to store containers, a higher speed factor (fewer container reads) represents a higher restore performance. The high performance storage scenario (e.g., SSD) is not considered in this paper.

6.1 Experimental Setup and Data Sets

The prototype is deployed on a Dell PowerEdge R430 server with a Seagate ST1000NM0033-9ZM173 SATA hard disk of 1TB capacity as the storage. The server has a 2.40GHz Intel Xeon with 24 cores and 64GB of memory. In our experiments, the container size is set to 4MB. To fairly compare the performance of these four deduplication engines, we try to use the same amount of memory in each engine. That is, for Capping and FCRC, the size of each segment is fixed at 5 containers. For LBW, the total size of the recipe cache and the rewrite candidate cache size is also 5 containers. For FAA, the look-ahead window size is 8 containers and the forward assembly area size is also 8 containers. ALACC has a 4 container size FAA, a 4 container size chunk cache, and an 8 container size look-ahead window.

In the experiments, we use six deduplication traces from the File System and Storage Lab (FSL) [34, 35, 36] as shown in Table 1. The traces were collected by the File Systems and Storage Lab (CS Department, Stony Brook University) and its collaborators. These traces cover two types of file system snapshots: MacOS and Homes. The former was collected on a Mac OS X Snow Leopard server running in an academic computer lab which contains SMTP, MySQL, HTTP, FTP, Wiki, etc., the latter contains snapshots of students' home directories from a shared network file system of the File System and Storage Lab. MAC1, MAC2, and MAC3 are three different backup traces from MacOS. FSL1, FSL2, and FSL3 are from FSL /home directory snapshots from 2014 to 2015.

Table 1: Characteristics of datasets

Dataset	MAC1	MAC2	MAC3	FSL1	FSL2	FSL3
TS(TB)[1]	2.04	1.97	1.97	2.78	2.93	0.63
US(GB)[2]	121.8	134.5	142.3	183.7	202	71
ASR(%)[3]	45.99	46.50	46.80	42.49	39.96	33.60
SV(%)[4]	99.28	96.60	95.48	97.87	98.64	93.00
IV[5]	5	20	60	5	20	60

[1] TS stands for the total data size of the trace.
[2] US stands for the unique data size of the trace.
[3] ASR stands for the average self-referenced chunk ratio in each version.
[4] SV stands for the average similarity between each version.
[5] IV stands for the time interval (days-based) between each version.

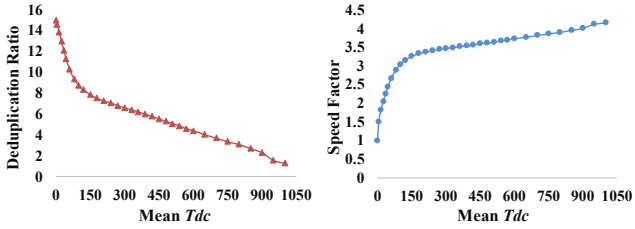
Each trace has 10 full backup snapshots and we choose the version with the 4KB average chunk size. As shown in SV (average similarity between versions) of Table 1, in each trace at least 93% of the data chunks appears in the previous versions. This indicates that most data chunks are duplicate data chunks and they are fragmented. The time interval (IV) between versions of each trace is also shown in Table 1. We select different IV for each trace such that the fragmentation and version similarity are varied in different traces. Each trace record contains the chunk hash value (17 bytes), chunk size, and compression ratio. The implemented deduplication engines use the chunk size and hash value to simulate the deduplication process without the chunking and chunk hash generating steps. Before replaying a trace, all data in the cache is cleared. For each experiment, we run the traces 3 times and present the average value.

6.2 Influence of LBW Size and Mean Value of T_{dc}

In the LBW design, different configurations of LBW size and different mean values of CNRC threshold T_{dc} will influence the deduplication ratio and the speed factor. We use the MAC2 trace as an example and design a set of experiments to investigate these relationships. In the restore process, we use FAA as the restore caching scheme and set the FAA size to 8 containers. To investigate the influence of LBW size, we fix the mean T_{dc} to 120 by setting the two bounds while the size of the LBW changes from 16 to 2 containers. The results are shown in Table 2. As the LBW size decreases, the deduplication ratio decreases slightly. It indicates that, when the threshold configuration is the same, if we reduce the LBW size, less information is maintained in the window and more data chunks are selected to be rewritten. The deduplication ratio decreases even faster when the LBW size is smaller than the FAA size, which is caused by a large num-

Table 2: The change of the deduplication ratio (DR) and the speed factor (SF) with different LBW size and a fixed mean T_{dc} (mean T_{dc} =120)

LBW size	16	14	12	10	8	6	4	2
DR	8.53	8.53	8.53	8.50	8.45	8.37	8.27	8.07
SF	2.80	2.84	2.87	2.92	3.01	3.14	3.16	3.18



(a) The deduplication ratio variation (b) The speed factor variation

Figure 5: The change of deduplication ratio and speed factor with different mean T_{dc} and a fixed LBW size (LBW=5)

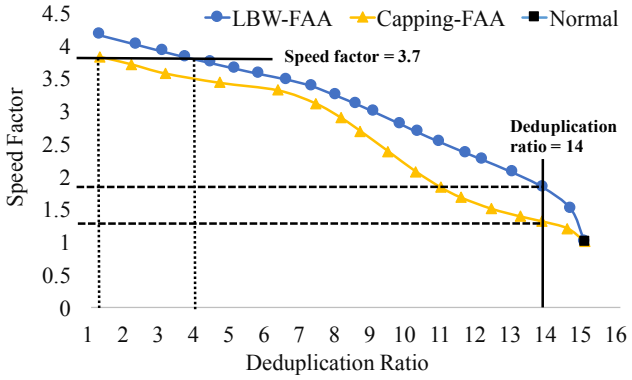


Figure 6: The speed factor of Capping and LBW when different deduplication ratios are achieved for the same trace. The Normal scheme is a black point in the figure

ber of unnecessary rewrites due to a small LBW size. In contrast, when the LBW size is close to or even smaller than the FAA size as shown in Table 2, there is a relatively large increase in the speed factor. Therefore, to achieve a good tradeoff between the deduplication ratio and the speed factor, the LBW size should be compatible with the FAA size, which is the cache-effective range in this case.

In another experiment, we fix the LBW size to 8 containers, FAA size to 8 containers, and vary T_{dc} from 0 to 1000. With a higher T_{dc}, more old containers will be rewritten since it is harder for each old container to have more than T_{dc} duplicate data chunks in the LBW. Therefore, as shown in Figure 5(a), the deduplication ratio decreases as T_{dc} increases. Also, the speed factor increases since more data chunks are rewritten as shown in Figure 5(b). When T_{dc} is zero, there is no rewrite and the deduplication regresses to the Normal deduplication case. On the other hand, when T_{dc}= 1000 (one container stores 1000 data chunks on average), almost every duplicate chunk is re-written. Thus, no deduplication is performed (i.e., deduplication ratio is 1) and the speed factor is 4. Note that, the decreasing of the deduplication ratio as well as the increasing of the speed factor are at a faster pace when T_{dc} is less than 100. When T_{dc} is larger than 100, the two curves vary linearly at a slower pace. It shows that we can achieve a good tradeoff between deduplication ratio reduction and restore performance improvement when T_{dc} is set smaller than 100 for this trace. The "knee" point can be different in other workloads. It is closely related to the CNRC distribution of old containers.

6.3 Deduplication Ratio & Speed Factor Comparison

In this subsection, we compare the performance of Capping with LBW in terms of deduplication ratio and speed factor when deduplicating the last version of MAC2. We choose FAA as the restore engine for Capping and LBW. In both cases, we set the FAA size to 8 containers. For Capping, the segment size is set to 5 containers and we run with the capping level varying from 0 to 70. For LBW, the memory for the rewrite candidate cache and the recipe cache is also 5 containers and the LBW size is fixed to 8 containers. We vary T_{dc} from 0 to 1000 by adjusting the targeted deduplication ratio and the targeted number of container reads. To fairly compare Capping and LBW, we need to compare the speed factor of the two designs when they have the same deduplication ratio.

The results are shown in Figure 6. We can clearly conclude that when the deduplication ratio is the same, the speed factor of LBW is always higher than that of Capping. For example, when the speed factor is 3.7, Capping has a deduplication ratio of 1.2, which indicates a very high rewrite number (close to no deduplication). With the same speed factor, LBW has a deduplication ratio of 4, which means only 25% of data chunks are stored. With the deduplication ratio of 14, Capping has a speed factor of 1.27 and LBW has a speed factor of 1.75 which is about 38% higher than that of Capping. If no rewrite is performed (Normal deduplication process), the deduplication ratio is 15 while the speed factor is close to 1. When the deduplication ratio increases from 8 to 14, which indicates fewer data chunks are rewritten and is close to the no rewrite case (Normal), the speed factor of LBW ranges from 20% to 40% higher than that of Capping. In general, LBW can achieve a better tradeoff between deduplication ratio reduction and speed factor improvement especially when the deduplication ratio is high.

6.4 Restore Performance Comparison

To obtain the performance of Normal, Capping, FCRC, and LBW as deduplication engines, we pair them with two restore engines (FAA and ALACC). Since a small reduction of deduplication ratio (i.e., fewer data chunk rewrites)

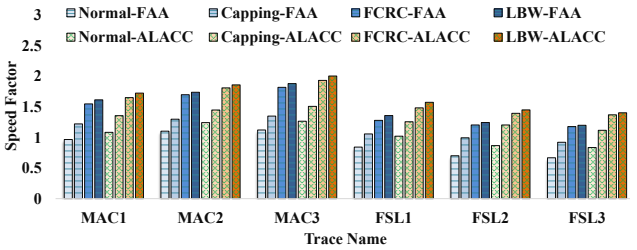
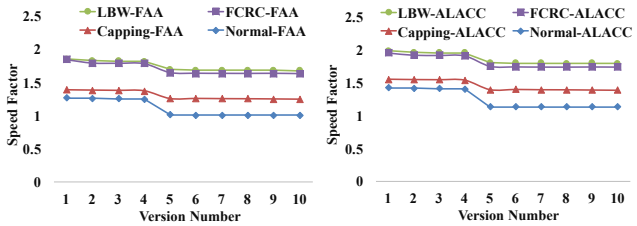


Figure 7: The mean speed factor of Normal, Capping, FCRC, and LBW for the 10 versions of six traces when restoring with FAA and ALACC



(a) The speed factor of using FAA as the restore caching scheme

(b) The speed factor of using ALACC as the restore caching scheme

Figure 8: The speed factor comparison of 8 combinations of deduplication-restore experiments when deduplicating and restoring the trace MAC2

is preferred in production, we configure the parameters of Capping, FCRC, and LBW in the following experiments to achieve the deduplication ratio reduction of 7%.

Each trace has 10 backup versions and the mean speed factors achieved by all 8 combination schemes are shown in Figure 7. If data chunk rewrite is not applied (Normal design) in the deduplication process, even with FAA or ALACC as the restore caching schemes, the speed factor is always lower than 1.3. In three FSL traces, the speed factor is even lower than 1, which means reading out a 4MB size container cannot even restore 1MB of data. It indicates that the data chunks are fragmented and the two caching schemes cannot effectively reduce a large number of container reads. By rewriting duplicate data chunks, the speed factor is improved. For all 6 traces, we find that with the same deduplication ratio, LBW always achieves the highest speed factor (best restore performance) when the same restore engine is used. The speed factor of FCRC is lower than that of LBW, but higher than that of Capping. Due to a lower average self-reference count ratio (ASR) shown in Table 1, the duplicate data chunks of FSL traces are more fragmented than that of MAC traces. Thus, the overall speed factor of FSL traces is lower than those of MAC traces. In general, when using the same restore caching scheme, the speed factor of LBW is up to 97% higher than that of Normal, 41% higher than that of Capping, and 7% higher than that of FCRC.

We also compared the speed factors of different backup versions in trace MAC2. The evaluation results of using FAA and ALACC as the restore caching schemes are shown in Figure 8(a) and Figure 8(b), respectively. Among the 10 versions, the duplicate data chunks are more fragmented after version 4. Therefore, there is a clear speed factor drop at version 5. As shown in both figures, by using FCRC, the speed factor is improved a lot compared with the Normal and Capping schemes. LBW further improves the speed factor when compared with FCRC by addressing the rewrite accuracy issues caused by a fixed segment partition and considering the cache-effective range as well as container read efficiency. So the speed factor of LBW is always the highest in all 10 versions. In general, with the same deduplication ratio, LBW can effectively reduce the number of container reads such that its restore performance is always the best.

7 Conclusion and Future Work

Rewriting duplicate data chunks during the deduplication process is an important and effective way to reduce container reads, which cannot be achieved by caching schemes during restore. In this paper, we discuss the limitations of capping design and propose the FCRC scheme based on capping to improve rewrite selection accuracy. To further reduce the inaccurate rewrite decisions caused by the fixed segment partition used by capping and FCRC, we propose the sliding look-back window based rewrite approach. By combining the look-back mechanism and the caching effects of the restore process, our approach makes better tradeoffs between the reduction of deduplication ratio and container reads. In our experiments, the speed factor of LBW is always better than that of capping and FCRC when the deduplication ratio is the same. In our future work, the adaptive LBW size and more intelligent rewrite policies will be investigated. Also, the garbage collection of the deleted data chunks will be investigated together with the rewrite design.

Acknowledgments

We thank all the members in CRIS group for providing the useful comments to improve our design. We would like to thank our shepherd, Keith Smith, for his useful comments, suggestions, and help in the paper revision. This work was partially supported by NSF awards 1421913, 1439622, 1525617, and 1812537.

References

[1] Robert E Fontana, Gary M Decad, and SR Hetzler. The impact of areal density and millions of square inches (MSI) of produced memory on petabyte shipments of TAPE, NAND flash, and HDD storage class memories.

In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST 13)*, pages 1–8. IEEE, 2013.

[2] https://www.backblaze.com/blog/hdd-vs-ssd-in-data-centers/.

[3] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *6th USENIX Conference on File and Storage Technologies (FAST 08)*, pages 1–14, 2008.

[4] Michal Kaczmarczyk, Marcin Barczynski, Wojciech Kilian, and Cezary Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *Proceedings of the 5th Annual International Systems and Storage Conference, Haifa, Israel (SYSTOR 12)*, pages 1–12, 2012.

[5] Youngjin Nam, Guanlin Lu, Nohhyun Park, Weijun Xiao, and David HC Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *2011 IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 581–586. IEEE, 2011.

[6] Young Jin Nam, Dongchul Park, and David HC Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 12)*, pages 201–208. IEEE, 2012.

[7] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 181–192, 2014.

[8] Mark Lillibridge, Kave Eshghi, and Deepavali Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 183–198, 2013.

[9] Bo Mao, Hong Jiang, Suzhen Wu, Yinjin Fu, and Lei Tian. SAR: SSD assisted restore optimization for deduplication-based storage systems in the cloud. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage (NAS 12)*, pages 328–337. IEEE, 2012.

[10] Zhichao Cao, Hao Wen, Fenggang Wu, and David HC Du. ALACC: accelerating restore performance of data deduplication systems using adaptive look-ahead window assisted chunk caching. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 309–323. USENIX Association, 2018.

[11] Wei Zhang, Tao Yang, Gautham Narayanasamy, and Hong Tang. Low-cost data deduplication for virtual machine backup in cloud storage. In *5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 13)*, 2013.

[12] Fanglu Guo and Petros Efstathopoulos. Building a high-performance deduplication system. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*, pages 271–294, 2011.

[13] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan. Design tradeoffs for data deduplication performance in backup workloads. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 331–344, 2015.

[14] Jaehong Min, Daeyoung Yoon, and Youjip Won. Efficient deduplication techniques for modern backup operation. *IEEE Transactions on Computers*, 60(6):824–840, 2011.

[15] Deepavali Bhagwat, Kave Eshghi, Darrell DE Long, and Mark Lillibridge. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *2009 IEEE 17th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 09)*, pages 1–9. IEEE, 2009.

[16] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *1st USENIX Conference on File and Storage Technologies (FAST 02)*, pages 89–101, 2002.

[17] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki. HYDRAstor: A scalable secondary storage. In *7th USENIX Conference on File and Storage Technologies (FAST 09)*, pages 197–210, 2009.

[18] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. iDedup: latency-aware, inline data deduplication for primary storage. In *10th USENIX Conference on File and Storage Technologies (FAST 12)*, pages 1–14, 2012.

[19] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, and Ming Zhao. Cached-edup: in-line deduplication for flash caching. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 301–314, 2016.

[20] Biplob K Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *2010 USENIX Annual Technical Conference (USENIX ATC 10)*, 2010.

[21] Yoshihiro Tsuchiya and Takashi Watanabe. DBLK: Deduplication for primary block storage. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST 11)*, pages 1–5. IEEE, 2011.

[22] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. Dmdedup: Device mapper target for data deduplication. In *2014 Ottawa Linux Symposium*, 2014.

[23] Sonam Mandal, Geoff Kuenning, Dongju Ok, Varun Shastry, Philip Shilane, Sun Zhen, Vasily Tarasov, and Erez Zadok. Using hints to improve inline block-layer deduplication. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 315–322, 2016.

[24] Zhuan Chen and Kai Shen. OrderMergeDedup: Efficient, failure-consistent deduplication on flash. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 291–299, 2016.

[25] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. Primary data deduplication-large scale study and system design. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 285–296, 2012.

[26] J. bonwick. ZFS deduplication, November 2009. https://blogs.oracle.com/bonwick/zfs-deduplication-v2.

[27] Veritas netbackup cloud administrator's guide. https://www.veritas.com/content/support/en_us/doc/ka6j000000004pkaaq.

[28] Dongchul Park, Ziqi Fan, Young Jin Nam, and David HC Du. A lookahead read cache: Improving read performance for deduplication backup storage. *Journal of Computer Science and Technology*, 32(1):26–40, 2017.

[29] Michal Kaczmarczyk and Cezary Dubnicki. Reducing fragmentation impact with forward knowledge in backup systems with deduplication. In *Proceedings of the 8th Annual International Systems and Storage Conference, Haifa, Israel (SYSTOR 15)*, pages 1–12, 2015.

[30] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Jingning Liu, Wen Xia, Fangting Huang, and Qing Liu. Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge. *IEEE Transactions on Parallel and Distributed Systems*, 27(3):855–868, 2016.

[31] Yujuan Tan, Baiping Wang, Jian Wen, Zhichao Yan, Hong Jiang, and Witawas Srisa-an. Improving restore performance in deduplication-based backup systems via a fine-grained defragmentation approach. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[32] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. A cost-efficient rewriting scheme to improve restore performance in deduplication systems. In *2017 IEEE 33th Symposium on Mass Storage Systems and Technologies (MSST 17)*, 2017.

[33] Jie Wu, Yu Hua, Pengfei Zuo, and Yuanyuan Sun. Improving restore performance in deduplication systems via a cost-efficient rewriting scheme. *IEEE Transactions on Parallel and Distributed Systems*, 2018.

[34] http://tracer.filesystems.org/.

[35] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 261–272, 2012.

[36] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, et al. A long-term user-centric analysis of deduplication patterns. In *2016 IEEE 32nd Symposium on Mass Storage Systems and Technologies (MSST 16)*, pages 1–7. IEEE, 2016.