

# A Cost-efficient Rewriting Scheme to Improve Restore Performance in Deduplication Systems

Jie Wu   Yu Hua   Pengfei Zuo   Yuanyuan Sun

Wuhan National Laboratory for Optoelectronics

School of Computer, Huazhong University of Science and Technology, Wuhan, China

Corresponding author: Yu Hua (csyhua@hust.edu.cn)

**Abstract**—In chunk-based deduplication systems, logically consecutive chunks are physically scattered in different containers after deduplication, which results in the serious fragmentation problem. The fragmentation significantly reduces the restore performance due to reading the scattered chunks from different containers. Existing work aims to rewrite the fragmented duplicate chunks into new containers to improve the restore performance, which however produces the redundancy among containers, decreasing the deduplication ratio and resulting in redundant chunks in containers retrieved to restore the backup, which wastes limited disk bandwidth and decreases restore speed. To improve the restore performance while ensuring the high deduplication ratio, this paper proposes a cost-efficient submodular maximization rewriting scheme (SMR). SMR first formulates the defragmentation as an optimization problem of selecting suitable containers, and then builds a submodular maximization model to address this problem by selecting containers with more distinct referenced chunks. We implement SMR in the deduplication system, which is evaluated via two real-world datasets. Experimental results demonstrate that SMR is superior to the state-of-the-art work in terms of the restore performance as well as deduplication ratio. We have released the source code of SMR for public use.

## I. INTRODUCTION

Data deduplication has been widely used in backup systems to save storage space [1], [2], [3], [4], [5], [6]. It divides incoming data stream into small and variable pieces, called chunks [7], and identifies each chunk by its small-size signature, called fingerprint, via hash functions, such as SHA-1, SHA-256 and MD5 [7], [8]. A fingerprint index maps fingerprints of the stored chunks to their physical addresses. [9], [10] These chunks are stored into several large fixed-size storage units called containers, to preserve the spatial locality of the backup data stream [2], [9], [11], [12], [13]. A container is the basic unit of reads and writes. Based on the redundancy of incoming chunks, a data deduplication system needs to carry out different operations. Specifically, duplicate chunks are replaced with the references to existing identical copies stored in old containers, and unique chunks are written into new containers. Each backup has a recipe to record the references to the containers of each backup chunk. In a restore phase, the restore algorithm scans down the recipe of the backup to determine which containers need to be retrieved from disks to the restore cache, which contains the prefetched containers, to restore the target chunks of the backup stream.

Although data deduplication is space-efficient, logically consecutive chunks have to be physically scattered in different containers, thus causing chunk fragmentation [9], [11], [14], [15], [16]. The fragmentation severely degrades the restore performance, while the infrequent restore is very important and becomes the main concern from users [17]. First, original sequential disk accesses of reading logically consecutive chunks become many random ones, while random accesses perform poorly in disks due to the penalty of disk seeks [11], [18]. Second, due to the fragmentation, some containers, containing a few referenced chunks (defined as the chunks referenced by the backup), are retrieved to restore the backup. The remaining unreferenced chunks in the retrieved containers are not accessed by the data stream, thus causing the waste of limited disk bandwidth and decreasing restore speed.

To address the fragmentation problem, several schemes propose rewriting algorithms to rewrite fragmented duplicate chunks during the backup, such as Capping [11] and NED [12]. They aim to obtain a suitable trade-off between deduplication efficiency and restore performance via selective deduplication upon containers with more referenced chunks.

We observe that with the increase of the number of backup versions, more and more duplicate chunks are rewritten into new containers, as described in Section II. Thus, multiple identical copies of these chunks are stored in different containers, causing redundancy among containers. However, the containers selected by existing schemes [11], [12] are suboptimal due to overlooking the redundancy among the containers. Hence, some containers with many redundant chunks, which are not referenced by the backup, are selected, thus wasting some disk bandwidth and slowing down the restore speed.

To address this problem, we propose a submodular maximization rewriting scheme (SMR) to efficiently select containers for deduplication, and judiciously rewrite duplicate fragmented chunks. SMR aims to select a limited number of containers which offer more distinct referenced chunks for the backup, thus reducing the number of redundant or unreferenced chunks in the selected containers, to gain a better trade-off between deduplication efficiency and restore performance. If more distinct referenced chunks are offered for the current backup stream, more chunks in the backup stream can be deduplicated, thus saving storage space. Since less

chunks are rewritten into new containers, SMR decreases the number of containers retrieved for restore. In addition, SMR also reduces the disk accesses of redundant and unreferenced chunks fetched in the restore phase. Hence, SMR achieves better restore performance as well as deduplication ratio.

In summary, the paper makes the following contributions.

- We observe that due to overlooking the redundancy among containers, existing solutions [11], [12] potentially choose suboptimal containers with many redundant chunks, which decreases the restore performance. This is an important problem for improving entire system performance.
- We propose a submodular maximization rewriting scheme, i.e., SMR, to select containers with more distinct referenced chunks for the backup, reducing the waste of disk accesses caused by redundant and unreferenced chunks in the restore phase. It can deduplicate more chunks and rewrite less chunks, and less containers are retrieved during the restore, gaining a better trade-off between deduplication efficiency and restore performance.
- We implement our scheme in the deduplication system and evaluate the performance via two real-world backup datasets. Compared with the state-of-the-art schemes [11], [12], experimental results demonstrate that our scheme obtains higher deduplication ratio as well as restore performance. We have released the source code of SMR for public use at <https://github.com/courageJ/SMR>.

The rest of this paper is organized as follows. In Section II, we describe the background and motivation. We present the design of SMR and evaluation respectively in Sections III and IV. Related work is discussed in Section V. We conclude our paper in Section VI.

## II. BACKGROUND AND MOTIVATION

### A. Chunk Fragmentation

Data deduplication saves storage capacity by storing one copy of the duplicated data. But the logically contiguous chunks of each backup are scattered all over different containers, called chunk fragmentation [9], [11], [14], [15], [16]. Thus the restore of each backup needs many random I/Os to the containers, which perform poorly in disks. Hence, chunk fragmentation significantly decreases the restore performance.

We simulate the baseline deduplication without rewriting on two real-world datasets, including GCC [19] and Linux [20] (detailed in Section IV) to explore the degree of chunk fragmentation and the restore performance. The chunk fragmentation level (CFL) [15], [16] is a quantitative metric to measure the level of chunk fragmentation per data stream, which is defined as a ratio of the optimal number of containers without any deduplication scheme with respect to the number of containers after deduplication to store the backup data stream. The CFL ranges from 0 to 1. The smaller CFL indicates that the physical distribution of the data stream is more scattered, resulting in lower restore speed. The speed factor [11] (detailed in Section IV) is a metric to evaluate

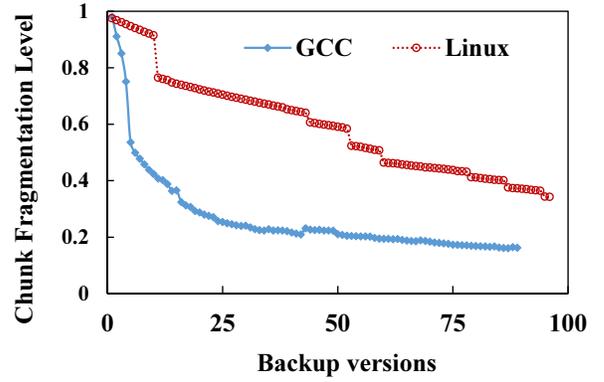


Fig. 1. The chunk fragmentation over time.

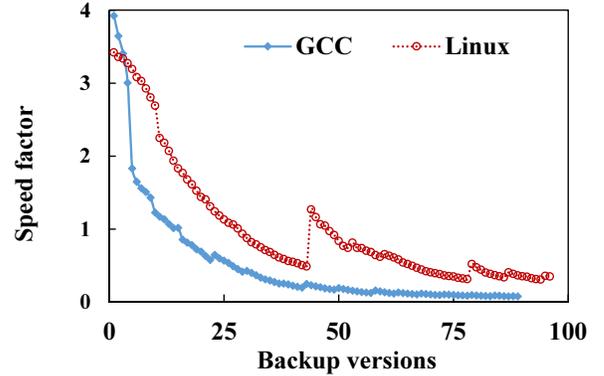


Fig. 2. The speed factor over time.

the restore performance. The higher speed factor indicates the better restore performance. As shown in Figure 1, the fragmentation level is mainly in a downward trend, indicating that the physical dispersion becomes more severe. As shown in Figure 2, the speed factor declines severely, indicating the decrease of restore performance. A few exceptions in the Linux datasets are the major revision updates, which have more new data stored consecutively. When more backups arrive, more shared chunks appear, which exacerbates the backup fragmentation and decreases the restore performance. Hence, how to mitigate the large slowdown of restore performance over time caused by the increasing degree of chunk fragmentation is an important problem to be concerned.

### B. The Selective Container Deduplication Schemes

In order to address the fragmentation problem and improve restore performance, some schemes about selective container deduplication have been proposed, e.g., Capping [11] and NED [12]. They define the fraction of chunks referenced by the backup in a container as the container's utilization and try to select containers with higher utilization for a backup to deduplicate. These schemes rewrite the duplicate chunks which refer to lower-utilization containers, to alleviate the waste of disk bandwidth caused by the unreferenced chunks in these containers in the restore phase. Hence, the restore

performance has been improved with the cost of decreasing storage efficiency.

Specifically, Capping splits the backup data stream into fixed-length segments, and uses a buffer to temporarily store them. The fingerprint index is further queried to conjecture which containers can be referenced by the chunks in the current segment. In fact, this is a trade-off between deduplication ratio and restore performance by setting the amount limit  $CAP\_T$  of selected containers for deduplication. If the amount  $N$  of the containers which contain referenced chunks for the current segment is more than  $CAP\_T$ , the top  $CAP\_T$  containers are selected and used in deduplication according to their utilization. The chunks of the segment are deduplicated if having identical copies in the selected containers. Otherwise, they are rewritten into new containers. Capping improves restore performance by limiting the amount of containers that a segment can refer to, and selects the containers with higher utilization. Moreover, NED computes the ratio of the sum size of referenced chunks to that of the stored chunks in each container for the current backup segment. If the ratio of a container is lower than a threshold  $NED\_T$ , the chunks in the segment that can refer to this container are regarded as fragmented chunks, which are further rewritten into new containers. NED aims to select some containers with utilization over the threshold for deduplication, which improves restore performance and mitigates data fragmentation.

### C. Problem Statement

Multiple rewriting schemes, e.g., Capping [11] and NED [12], have been proposed to rewrite fragmented duplicate chunks to improve restore performance. However, with the increase of the number of backup versions, more and more duplicate chunks are rewritten into new containers. In consequence, many identical chunks are stored in different containers, thus increasing the redundancy among containers. Due to overlooking the redundancy among containers, existing schemes [11], [12] need to count some redundant chunks, which are not referenced by the backup, in computing the utilization of containers. Hence, some suboptimal containers are selected for the backup, which exacerbates disk accesses and slows down the restore speed.

Specifically, there exist multiple identical chunks among the selected containers referenced by the backup. One of these identical chunks can be referenced to deduplicate and restore all chunks with identical context of the backup. Due to playing the same function, other redundant chunks are not needed to be referenced by the backup. These redundant chunks in the selected containers hence fail to be regarded as referenced chunks in computing the utilization for these selected containers that are mistakenly considered to achieve higher utilization.

We take Capping [11] as an example to show the effects of the redundancy among containers on the restore performance. As shown in Figure 3, three data streams with thirteen chunks are backed up and the amount of the selected containers in Capping,  $CAP\_T$ , is set as 2. The number of chunks stored

in each container is fixed as 5. The chunks in the first stream are stored in containers I, II, and III. When the second stream arrives, the utilizations are computed for each container: I is 5, II is 2 and III is 3. The containers I and III are the top 2 selected containers. Two chunks F and G referring to the container II and the remaining three unique chunks are written into the new container IV. Moreover, when the third stream arrives, the utilizations for each container are also computed: I is 3, II is 5, III is 0 and IV is 4. Thus top 2 selected containers are II and IV. 7 chunks are deduplicated. 6 remaining chunks which do not refer to the two containers are stored in new containers. As shown in the third stream backup, both containers II and IV have chunks F and G. Thus, in a restore phase, fetching the container IV only restores the chunks O and P, since the chunks F and G have been already restored by the container II. This scheme deduplicates less data due to selecting the container IV to deduplicate the chunks F and G that have been deduplicated by the container II. Thus, the chunks F and G in the container IV are considered to be referenced chunks when the Capping scheme counts the utilization for the container IV to select container IV, but the chunks F and G actually are not, wasting disk accesses. As shown in the example above, more backups lead to more rewritten chunks in the new containers. Hence, there are more identical chunks stored in different containers, increasing the redundancy among containers. Existing schemes [11], [12] determine a chunk as the referenced chunk as long as it can be referenced by the backup. But considering all chunks in the selected containers, some chunks are redundant to be referenced, leading to extra disk accesses in a restore phase.

### D. Observations and Motivations

1) *The Redundancy Among Containers:* More and more chunks are rewritten into new containers in a backup for better restore performance, and thus multiple copies are stored in different containers for identical chunks, increasing the redundancy among containers. We explore the redundancy among containers in two real-world datasets, including GCC [19] and Linux [20] (detailed in Section IV). We use Capping [11] as the example of the rewriting algorithm to backup the consecutive versions of each dataset in two backup experiments. One is to backup the datasets under the constant Capping level, i.e.,  $CAP\_T$ . The other is to backup under different  $CAP\_T$  values. The default size of segment in Capping is 20MB, which is recommended in the Capping paper [11]. The constant Capping level in the first experiment is set as 20 containers per 20MB segment.

We classify the stored chunks into two categories: unique chunks (i.e., chunks only stored in one container) and redundant chunks (i.e., chunks with multiple copies stored in different containers). We count the amount of the two kinds of chunks respectively.

As shown in Figure 4, the amount of redundant chunks becomes far more than that of unique chunks. The reason is that original unique chunks become chunks with multiple copies since incoming identical chunks are rewritten into new

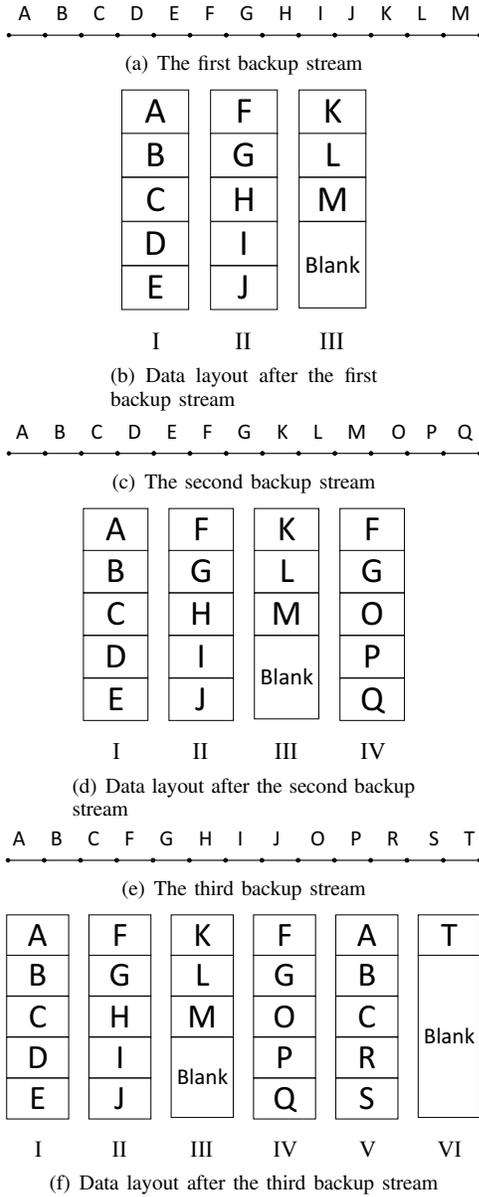
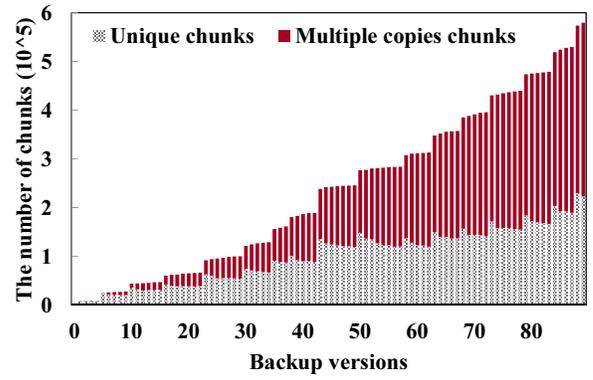


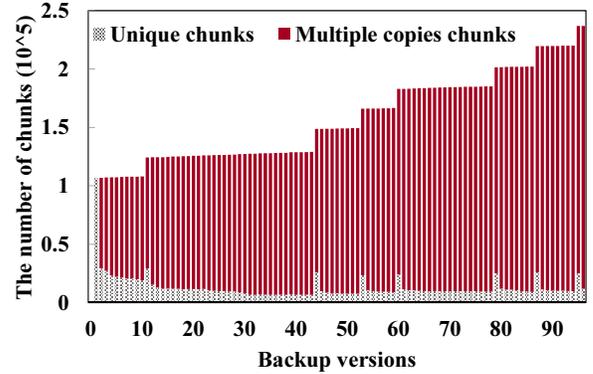
Fig. 3. An example of three consecutive backups with the Capping rewriting scheme.

containers. More and more containers have identical chunks. As shown in Figure 5,  $CAP\_T$  values vary from 5 to 40 per segment. Smaller  $CAP\_T$  indicates that less containers are selected for segments, achieving better restore performance, and more chunks are rewritten into new containers, which results in more redundant chunks. As shown in the two experiments, more and more chunks are rewritten into new containers to reduce data fragmentation, however increasing the redundancy among containers.

2) *Motivations*: Although existing selective container deduplication schemes improve restore performance by limiting the amount of the selected containers with less referenced chunks for deduplication, the redundancy among containers results in multiple identical chunks to be selected for deduplication and



(a) GCC



(b) Linux

Fig. 4. The redundancy among containers in consecutive versions backups with the Capping rewriting scheme.

fetches to restore the same chunks, wasting disk accesses. In essence, the wasted disk accesses in the restore phase are caused by selecting different containers with redundant chunks to deduplicate. Thus, when fetching containers to restore data stream, both unreferenced and redundant chunks in containers cause the waste of disk bandwidth and decrease the restore performance.

TABLE I  
THE AMOUNT OF DISTINCT REFERENCED CHUNKS FOR EVERY 2 CONTAINERS.

Container ID	Distinct Referenced Chunks	Chunks Amount
I, II	A B C F G H I J	8
I, III	A B C	3
I, IV	A B C F G O P	7
II, III	F G H I J	5
II, IV	F G H I J O P	7
III, IV	F G O P	4

We aim to improve the restore performance by selecting a subset of containers with more distinct referenced chunks for deduplication under the limits of selected containers to reduce the number of unreferenced and redundant chunks in the selected containers. For example, we perform our selection strategy on the three backup streams as shown in Figure 3. The amount of the selected containers is also set as 2, which is the same as Capping level, i.e.,  $CAP\_T$  in Figure 3. The

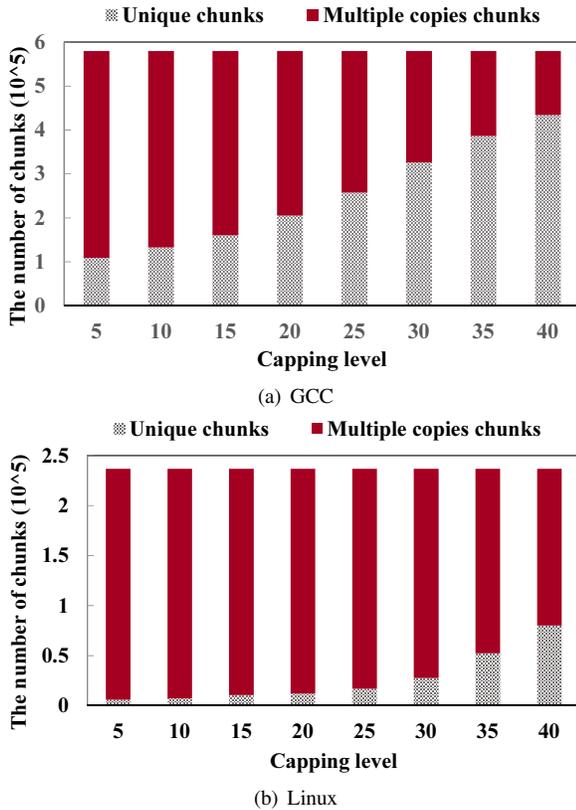


Fig. 5. The observations of redundancy among containers in various CAP\_T levels.

backup processes and containers distributions after the first two stream backup are the same as those in Figure 3. For the third data stream, when selecting 2 containers to deduplicate, we first count the amount of distinct referenced chunks for every 2 containers, as shown in Table I. The number of distinct referenced chunks in the subset, consisting of the containers I and II, is more than any other subsets. Selecting the containers I and II can deduplicate 8 chunks and write 5 chunks. The distributions of containers after the third backup stream are shown in Figure 6.

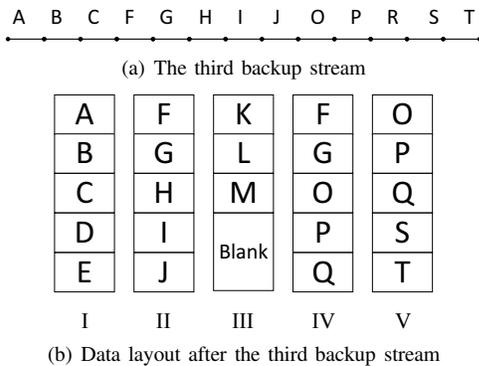


Fig. 6. The distributions of containers after the third backup.

Compared with Capping in Figure 3, we deduplicate 8 chunks A, B, C, F, G, H, I and J while Capping deduplicates

7 chunks F, G, H, I, J, O and P. We rewrite 2 duplicate fragmented chunks O and P into new containers while Capping rewrites 3 duplicate fragmented chunks A, B and C. Hence, we deduplicate more chunks and rewrite less chunks, saving more storage space. In the restore phase, we only fetch three containers I, II, V to restore the third data stream while Capping needs to retrieve four containers II, IV, V, VI. The reason is that we select two containers I and II, which have the largest number of distinct referenced chunks and no redundant chunks. Capping selects containers II and IV, which have higher utilizations and however contain two redundant chunks. The redundancy causes the actual number of referenced chunks in container IV to be much smaller than that in container I. Capping mistakenly considers the utilization of container IV to be the one of top 2 containers, which is selected for deduplication. Therefore, selecting containers with more distinct referenced chunks can deduplicate more chunks, rewrite less chunks and reduce the number of redundant and unreferenced chunks in the selected containers, achieving higher deduplication ratio and also improving restore performance.

### III. THE DESIGN OF SMR

#### A. An Architectural Overview

To reduce data fragmentation, our proposed Submodular Maximization Rewriting Scheme (SMR) selectively rewrites some fragmented duplicate chunks into new containers, and deduplicates the remaining duplicate chunks. SMR aims to trade the slight decrease of deduplication ratio for the high restore performance via efficiently selecting a limited number of old containers with more distinct referenced chunks for deduplication by a submodular maximization model. Specifically, in an old container, if there are many unreferenced and redundant chunks which are not needed to be referenced by the backup, there are few referenced chunks in the container. Thus, SMR rewrites the few referenced chunks of the old containers to reduce disk accesses caused by the unreferenced and redundant chunks in the restore phase. In a backup phase, there are many old containers sharing the duplicate chunks with the backup data stream. How to select the containers to perform deduplication is a trade-off between the restore performance and the deduplication ratio. Hence, SMR selects a suitable subset of containers to obtain better restore performance while ensuring the high deduplication ratio.

Figure 7 illustrates the architecture of SMR in a deduplication system. In the main memory, the system splits the input data streams into chunks and uses hash functions to identify them, then the chunks are grouped into segments. For each segment, after redundancy identification, in order to gain a suitable trade-off between deduplication ratio and restore performance, SMR determines which chunks are deduplicated or rewritten according to the redundancy information of segments. Finally, these rewritten and unique chunks are stored into the container pool. Meanwhile, the index and recipe are updated according to the fingerprints and addresses of these chunks.

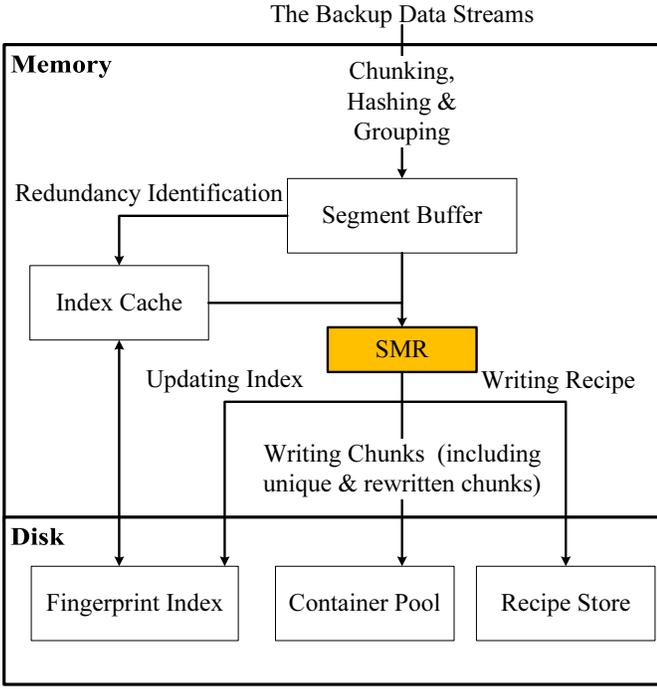


Fig. 7. The SMR Architecture.

Specifically, before a backup, the SMR level, i.e.,  $T$  old containers selected for the deduplication of every  $S$  MB segment, is configured and the backup stream is split into chunks, which are further grouped into segments. For each segment, we first read each chunk of the segment, and then determine which chunks have identical copies stored in containers and their located container IDs by inquiring the fingerprint index. In detail, the complete fingerprint index is stored on disks while the hot part is stored in memory to accelerate the indexing of fingerprints. Afterwards, we select  $T$  old containers with the largest number of distinct referenced chunks to perform deduplication. Finally, for each chunk in the segment, if finding an identical copy in the selected containers, the chunk is deduplicated. Otherwise, it is treated as new chunks to be stored in new containers.

### B. The Submodular Maximization Rewriting Scheme

In a restore phase, the containers containing the target chunks of the restored data stream are read from disks to the memory. One way to improve restore performance is to reduce the number of the retrieved containers.

In order to achieve this goal, SMR limits the number of old containers that referenced by the backup and reduces the amount of new containers storing the rewritten and unique chunks. Limiting the number of old containers means selecting a subset of old containers for the backup in deduplication. Reducing the amount of new containers means reducing the number of rewritten chunks, which depends on the old containers selection. Considering that the unreferenced and redundant chunks in old containers waste disk accesses in a restore phase, if a set of old containers with more distinct referenced chunks

are selected for the backup to perform deduplication, we are able to deduplicate more chunks and reduce the number of the containers to be rewritten into new containers. Thus, the number of the retrieved containers is reduced while alleviating the waste of disk accesses caused by unreferenced and redundant chunks, improving the restore performance.

We formulate the problem of container selection as a *subset selection problem*.

*Definition 1: (Subset Selection Problem)* Given a set of old containers to be selected  $V = (C_1, C_2, \dots, C_{|V|})$  and a budget  $T$ , i.e., the amount of selected containers, we aim to find a container subset  $S \subseteq V$ ,  $|S| \leq T$ , which can offer the largest number of distinct referenced chunks for the backup to perform deduplication under the constraints of the budget amount.

Here, we set the SMR level metric as the amount budget to limit the amount of old containers to be selected, which constrains the number of containers that need to be read in the restore phase to accelerate restore speed. The number of deduplicated chunks depends on the number of distinct referenced chunks in the selected containers. Hence, the container subset with the largest number of distinct referenced chunks can deduplicate the maximum number of chunks, and new containers ideally store the minimum number of rewritten chunks, which decreases the storage consumption. Moreover, we determine which subset of containers can be selected, rather than determining whether to be selected for each container, thus preventing redundant chunks from being counted into the number of the referenced chunks.

To address the problem of container subset selection, we build a submodular maximization model [21]. We first design a scoring function  $F : 2^V \rightarrow \mathbb{R}$  to indicate the amount of distinct referenced chunks in a subset. Thus the subsets offering more distinct referenced chunks are mapped to higher scores and the subsets offering less distinct referenced chunks are mapped to lower scores. The subset selection can be performed by the following computation:

$$S^* \in \operatorname{argmax}_{S \subseteq V} F(S) \quad s.t. |S| \leq T.$$

In order to describe the distinct referenced chunks in containers, the scoring function  $F$  is designed as:

$$F(S) = \left| \bigcup_{C_i \in S} w(C_i) \right|.$$

Specifically,  $w(C_i)$  represents all referenced chunks in the container  $C_i$ .  $\bigcup$  denotes the union of a collection of sets, indicating the set of all distinct elements in the collection. The  $|\cdot|$  denotes the number of all elements in the set. Hence,  $F(S)$  denotes the number of all distinct referenced chunks that all containers  $C_i \in S$  can offer.

In general, for arbitrary set functions, computing  $S^*$  is intractable [22]. Moreover, in the subset selection problem, there are  $\binom{N}{M}$  possible cases of selecting  $M$  containers from  $N$  containers, which is exponentially large for any reasonable  $M$  and  $N$  [23]. Hence, it is inefficient to compute all possible

cases. However, the maximization for any monotone submodular functions under some constraints can be efficiently solved by the greedy algorithm in a constant-factor mathematical quality guarantee [24]. The scoring function  $F$  is exactly a monotone submodular function, which is proved below. Thus, we can address the subset selection problem efficiently via the greedy algorithm [21].

We first introduce the definition and property of submodular functions [25].

**Definition 2: (Submodular)** Given a finite set, a set function  $f : 2^V \rightarrow \mathbb{R}$  that maps subset  $S \subseteq V$  of a finite ground set  $V$  to real numbers, is submodular if it satisfies: for any  $S \subseteq T \subseteq V$  and  $a \in V \setminus S$ ,

$$f(S \cup \{a\}) - f(S) \geq f(T \cup \{a\}) - f(T).$$

This states that the incremental benefit of adding an element to a smaller set is not less than that of adding an element to a larger set.

**Definition 3: (Monotone)** A set function  $f : 2^V \rightarrow \mathbb{R}$  is monotone if for every  $S \subseteq T \subseteq V$ ,  $f(S) \leq f(T)$ .

Based on the definition and property, we prove the scoring function  $F$  is a monotone submodular function. The operation  $\cap$  in the formulation, e.g.  $A \cap B$  denotes the intersection of two sets  $A$  and  $B$ , indicating a set that contains all elements of sets  $A$  and  $B$ .

**Statement 1:** The scoring function  $F(S) = |\bigcup_{C_i \in S} w(C_i)|$  is submodular.

*Proof 1:* Given any set  $S \subseteq T \subseteq V$  and element  $C_a \in V \setminus S$ , we have

$$F(S \cup \{C_a\}) - F(S) = |w(C_a)| - \left| w(C_a) \cap \left( \bigcup_{C_i \in S} w(C_i) \right) \right|,$$

indicating the number of chunks in container  $C_a$  but not in any containers  $C_i \in S$ , and

$$F(T \cup \{C_a\}) - F(T) = |w(C_a)| - \left| w(C_a) \cap \left( \bigcup_{C_i \in T} w(C_i) \right) \right|,$$

indicating the number of chunks in container  $C_a$  but not in any containers  $C_i \in T$ . Thus we have

$$\begin{aligned} & F(S \cup \{C_a\}) - F(S) - (F(T \cup \{C_a\}) - F(T)) \\ &= \left| w(C_a) \cap \left( \bigcup_{C_i \in T} w(C_i) \right) \right| - \left| w(C_a) \cap \left( \bigcup_{C_i \in S} w(C_i) \right) \right| \\ &= \left| w(C_a) \cap \left( \bigcup_{C_i \in (T \setminus S)} w(C_i) \right) \right| \\ &\geq 0. \end{aligned}$$

This indicates the number of chunks in  $C_a$ , which are not in any containers  $C_i \in T \setminus S$ . The left expression is not less than 0 in any case. Specially, when the chunk intersection of containers  $C_a$  and  $C_i \in T \setminus S$  is empty, the left expression is equal to 0. Thus,

$$F(S \cup \{C_a\}) - F(S) \geq F(T \cup \{C_a\}) - F(T),$$

indicating  $F$  is submodular according to Definition 2. That means when adding the new container  $C_a$  to the smaller set  $S$ , the incremental number of distinct referenced chunks is not smaller than that of adding  $C_a$  to the larger set  $T$ .

**Statement 2:** The scoring function  $F(S) = |\bigcup_{C_i \in S} w(C_i)|$  is monotone.

*Proof 2:* Given any set  $S \subseteq T \subseteq V$ , we have

$$\begin{aligned} F(T) - F(S) &= \left| \bigcup_{C_i \in T} w(C_i) \right| - \left| \bigcup_{C_i \in S} w(C_i) \right| \\ &= \left| \bigcup_{C_i \in T \setminus S} \left( w(C_i) - w(C_i) \cap \left( \bigcup_{C_j \in S} w(C_j) \right) \right) \right| \\ &\geq 0. \end{aligned}$$

$F(T)$  is always not smaller than  $F(S)$ , indicating  $F$  is monotone according to Definition 3. That means the number of distinct referenced chunks in the larger set  $T$  is not smaller than that in the smaller set  $S$ . As shown above, the scoring function  $F$  is proved to be submodular and monotone.

We design a greedy algorithm to select the subset of containers with the largest number of distinct referenced chunks effectively. As shown in Algorithm 1, the algorithm sequentially finds a container  $c_i$  in the remaining set  $V \setminus S_i$  for each iteration. Moreover,  $c_i$  has the maximum quantity in offering the chunks which are different from chunks in  $S_i$ . Adding  $c_i$  to  $S_i$  offers more distinct referenced chunks for the segment than other containers. When adding the containers from the set  $V \setminus S_i$  to  $S_i$ , if the increased number of distinct referenced chunks is not larger than 0, the iteration will stop, preventing containers without any referenced chunks from being selected and reducing disk accesses.

---

#### Algorithm 1 Greedy Selection Algorithm

---

**Input:** A set of containers to be selected:  $V$ . Submodular monotone function:  $F(\cdot)$ . All chunks in each container:  $w(\cdot)$ . The amount of containers selected:  $T$ .

**Output:** A set of containers  $S \subseteq V$ , where  $|S| \leq T$ .

```

1:  $S_0 \leftarrow \emptyset, i \leftarrow 0$ 
2: while  $|S_i| \leq T$  do
3:   Choose  $c_i \in \operatorname{argmax}_{c_i \in V \setminus S_i} (F(S_i \cup \{c_i\}) - F(S_i))$ 
4:   if  $(F(S_i \cup \{c_i\}) - F(S_i) == 0)$  then
5:     Break
6:   end if
7:    $S_{i+1} \leftarrow S_i \cup \{c_i\}$ 
8:    $i \leftarrow i + 1$ 
9: end while
10: return  $S_i$ 

```

---

## IV. PERFORMANCE EVALUATION

### A. Experiment Setup

We configure our experimental environment by extending a real-world open-source deduplication system, i.e., Destor [26],

which has been used in multiple research schemes and systems [9], [27], [28]. To examine the restore performance of our scheme, we compare SMR with two state-of-the-art selective rewriting schemes that leverage containers in deduplication systems, i.e., Capping [11] and NED [12], described in Section II.

We implement our scheme in Destor [26] on the CentOS operating system running on a 4-core Intel E5620 2.40GHz system with 24GB memory and 1TB hard disk. Due to the lack of source code of Capping, we faithfully implement its idea and main components described in its paper [11], which is also released in our public source code.

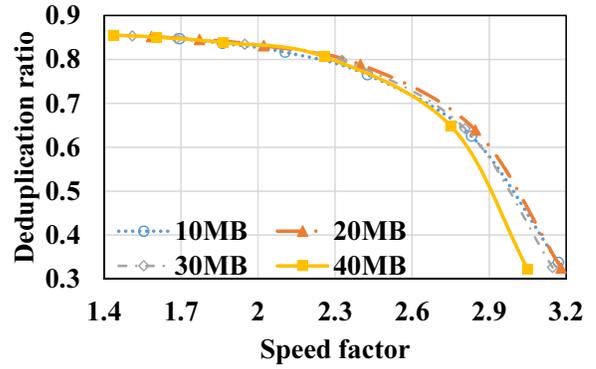
Two real-world datasets, including Linux and GCC, are used for evaluation, which have been evaluated for deduplication in the storage community [9], [29], [30], [31]. Specifically, Linux consists of 96 consecutive versions of unpacked Linux kernel sources from linux-4.0 to linux-4.7. The total size of the dataset is about 97GB. Moreover, GCC [19] consists of the source code of the GNU Compiler Collection. There are 89 consecutive versions from gcc-2.95 to gcc-6.1.0. The total size of the dataset is about 56GB.

In a deduplication system, each dataset is divided into variable-size chunks by using the content-based Rabin chunking algorithm [32]. The SHA-1 hash function [8] is used to generate the fingerprints of chunks. Since this paper mainly focuses on improving the restore performance rather than the fingerprint index access boosting, we simply store the complete fingerprint index in memory in these experiments. The restore cache stores the prefetched containers, in which the LRU replacement algorithm is used [11].

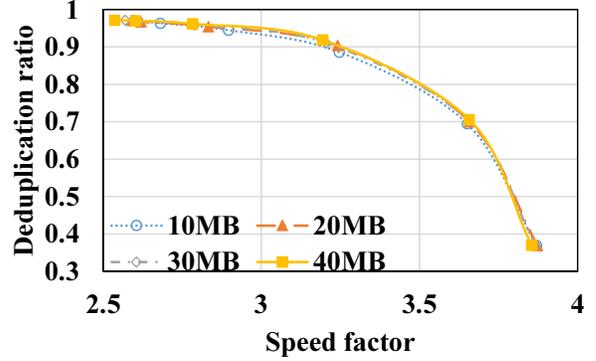
We use the speed factor [11] as the metric of the restore performance and the deduplication ratio and throughput to examine the deduplication efficiency. Specifically, speed factor [11] is to compute 1 divided by mean container read per MB of data restored, which is widely used to evaluate the restore performance [9], [11], [12]. Higher speed factor means that less containers are needed for the restored data per MB, thus indicating better restore performance. Moreover, deduplication ratio is the ratio of total size of the removed duplicate chunks to that of all backed up chunks. Furthermore, deduplication throughput is the amount of backed up data per second. The higher deduplication throughput indicates the faster backup speed.

### B. The Configurations of Parameters

1) *The Segment Size:* The chunks of a backup stream are grouped into fixed-sized segments in the backup and restore phases. The segment size sets the scale of fragment identification. Figure 8 shows the restore performance of SMR under various segment sizes. We observe that increasing the segment size makes little difference to the speed factor. Larger segment sizes fail to produce better results than smaller ones. For example, the deduplication and restore performance when the segment size is set as 20MB are better than that of the 40MB size. Hence, the segment size is set as 20MB by default, which is also the recommended setting in Capping and NED.



(a) GCC



(b) Linux

Fig. 8. The effect of segment sizes varies from 10MB to 40MB on deduplication and speed for selected SMR ratios (SMR level to the segment size). SMR ratios are 5/10, 10/10, 15/10, 20/10, 25/10, 30/10 containers per MB.

2) *The Cache Size:* Due to restoring the target data stream, the corresponding containers are retrieved to restore cache from disks. We consider the restore cache size for temporarily storing the containers in the restore phase. Figure 9 shows the restore performance of SMR under various cache sizes. Specifically, SMR T stands for a SMR level of T containers for each segment. For example, SMR 10 indicates that for each segment, 10 containers are selected for deduplication. We observe that if the cache size continuously increases, the restore speed is faster. The reason is that larger cache sizes allow more containers to be stored in the cache to restore chunks, reducing the time overhead of fetching containers when the cache misses occur. In addition, the increasing trend of speed factor becomes slower. We argue that for a large proportion of backup segments, about 20 containers are sufficient to restore a segment. For most segments, when the cache size is over 20, all containers can be fetched into cache at once without replacement. But the speed factor still increases since the total size of containers needed by a few segments is larger than the cache size. Hence, we set the default cache size as the total size of 30 containers with LRU cache replacement scheme, which can cater to temporary store needs of most cases.

3) *The Container Size:* Figure 10 illustrates the restore speed and deduplication ratio results of SMR when the

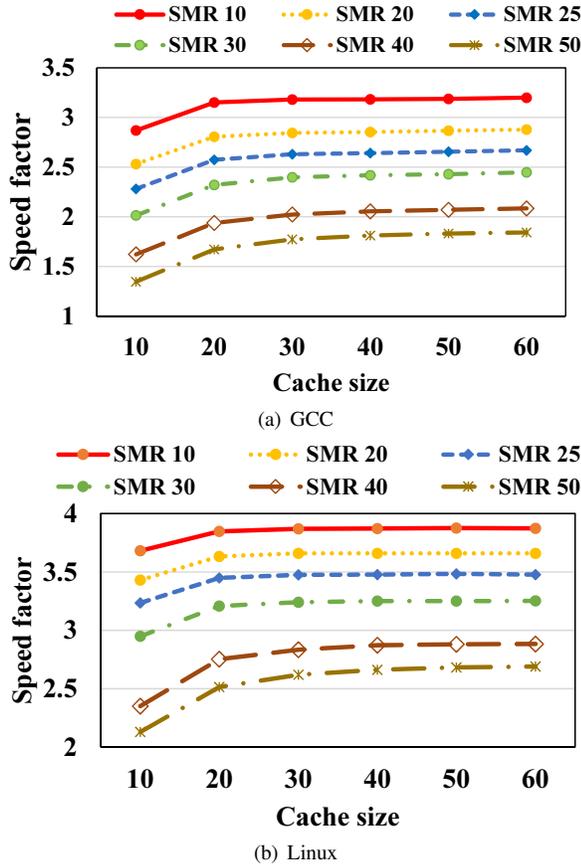


Fig. 9. The effect of various cache sizes on the restore performance. The cache size is the total size of # containers. SMR T denotes a SMR level of T containers per 20MB segment.

configured container sizes are set to be 1MB, 2MB, 4MB respectively. We use various SMR levels to conjecture the relationship between restore speed and deduplication ratio. Specifically, more containers can restore more chunks in one access, resulting in better restore performance under the same deduplication ratio. Hence, the default size of container is set as 4MB, which is also well-recognized in deduplication systems [2], [9], [11], [12], [26].

### C. Experimental Results and Analysis

SMR aims to obtain a suitable trade-off between deduplication efficiency and restore performance. We compare our scheme with two state-of-the-art schemes, i.e., Capping and NED. The experimental settings of Capping and NED faithfully follow the recommended parameters of their publications [11], [12]. For example, the segment size is set as 20MB. We evaluate the overall speed factor, deduplication throughput and deduplication ratio under different rewrite thresholds in SMR and the two compared schemes. Specifically, the Capping levels change from 5 to 60 containers per 20MB segment. The NED thresholds change from 0.1 to 1.0 per 20MB segment. Moreover, SMR levels vary from 5 to 60 containers per 20MB segment.

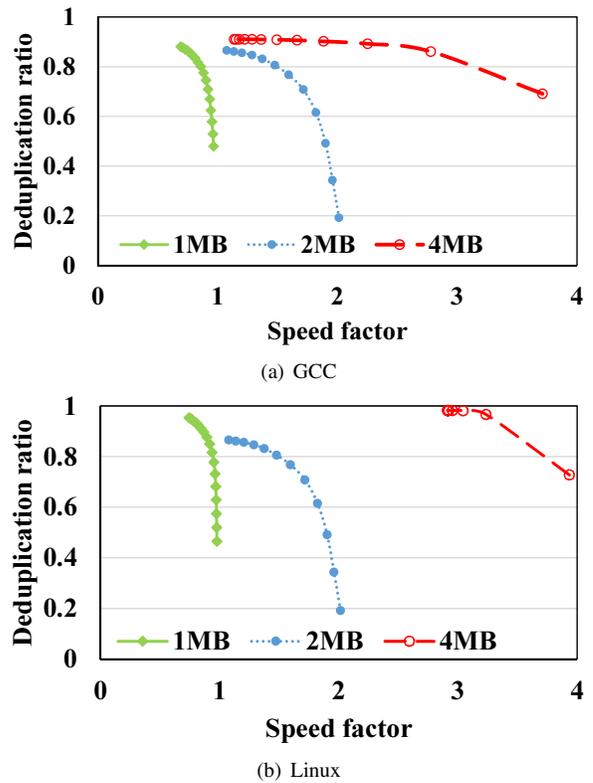


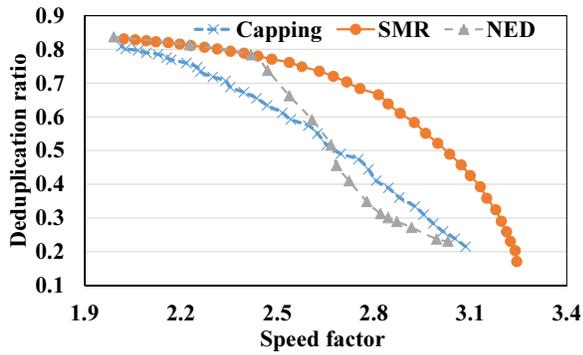
Fig. 10. The effect of various container sizes on the restore performance and speed factor. The container sizes are set to 1MB, 2MB, and 4MB respectively.

1) *The Relationship between Deduplication Ratio and Speed Factor:* In general, higher speed factor results in lower deduplication ratio, since more duplicate fragmented chunks are rewritten. As shown in Figure 11, the speed factors of SMR are larger than those of Capping and NED in the same deduplication ratios. Similarly, the deduplication ratios of SMR are larger than those of Capping and NED in the same speed factors. SMR can deduplicate more data while gaining better restore performance. Like our analysis in Section II, our scheme improves restore performance by reducing the amount of unreferenced and redundant chunks fetched in the restore phase. Thus, more chunks are deduplicated by more distinct referenced chunks in the selected containers. Less chunks are rewritten into new containers and less containers are needed to be retrieved during the restore.

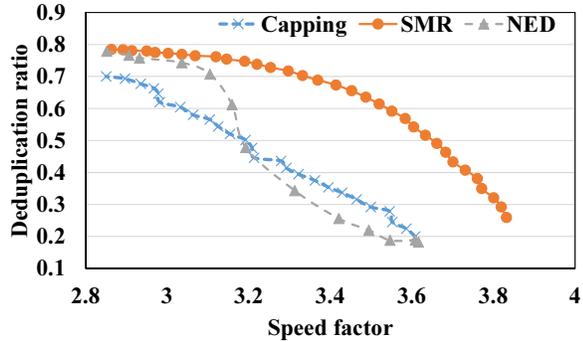
2) *Deduplication Throughput:* As shown in Figure 12, we compare the deduplication throughputs of SMR, Capping and NED under the same restore performance. In most cases, the deduplication throughputs of SMR are higher than those of Capping and NED.

### D. The SMR Level Settings

In our scheme, the SMR level, defined as the number of old containers selected for deduplication, i.e., is adjustable to improve restore performance to meet the needs of different cases. Figure 13 shows the effects of various SMR-level settings on the speed factor. The smaller SMR level results in

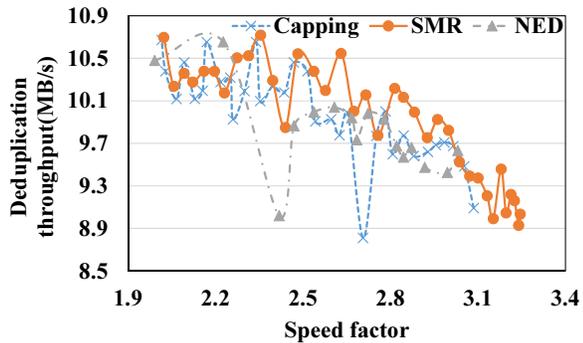


(a) GCC

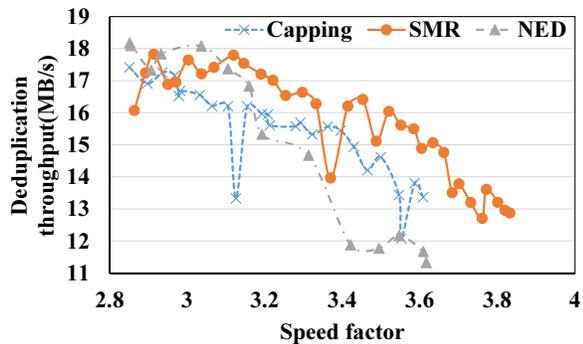


(b) Linux

Fig. 11. Deduplication ratios versus speed factors for various thresholds in the Capping levels, SMR levels and NED thresholds in two datasets.

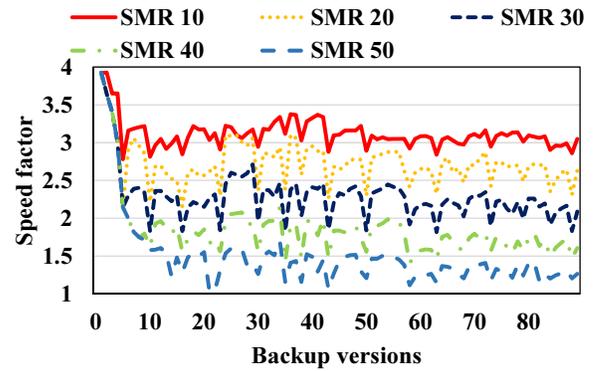


(a) GCC

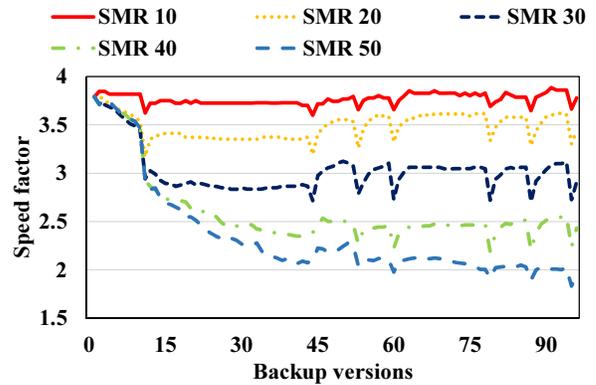


(b) Linux

Fig. 12. Deduplication throughput versus speed factor as Capping levels, SMR levels and NED thresholds vary in two data sets.



(a) GCC



(b) Linux

Fig. 13. The effects of various SMR levels on speed factor. SMR T denotes a SMR level of T containers per 20MB segment.

the higher speed factor. Because selecting less containers slows down the accumulation speed of data fragmentation. Less data fragmentation leads to better restore performance. Lower SMR level setting can alleviate the waste of disk accesses caused by unreferenced or redundant chunks in the retrieved containers. The SMR level is the key element to tradeoff deduplication efficiency and restore performance, and can be altered according to the demands of restore performance.

## V. RELATED WORK

In order to address the fragmentation problem, existing schemes mainly propose to leverage rewriting-based solutions to rewrite the duplicate fragmented chunks to mitigate the decrease of the restore performance, which is actually a tradeoff between deduplication ratio and restore performance.

In the primary storage deduplication, iDedup [18] deduplicates a sequence of chunks whose physical addresses are also sequential exceeding a minimum length threshold. POD [33] identifies capacity-insensitive and performance-sensitive small duplicate writes and files to further improve restore performance. Unlike them, SMR aims to improve restore performance in deduplication-based backup storage systems.

Nam et al. [15], [16] selectively deduplicate chunks with the proposed quantitative metric called chunk fragmentation level (CFL) for backup workloads. This scheme becomes inefficient to deduplicate chunks that are not included in the sequence

like former backups. The backup and restore units in SMR are segments. Due to overlooking the order of chunks in a segment, the chunks which are not in the sequence like former backups can still be deduplicated.

Context-Based Rewriting(CBR) [34] rewrites fragmented chunks by judging the degree of the difference between stream and disk contexts. It limits the entire amount of deduplication loss to a small value, which overlooks the unbounded fragmentation and decreases restore speeds. Capping [11] selectively deduplicates chunks referring to top T containers ordered by the number of referenced chunks in containers, and rewrites the remaining duplicate chunks to improve restore speed. NED [12] selectively rewrites fragmented chunks if the reference ratio of the referred storage is lower than a threshold. In fact, CBR, Capping and NED determine the fragmented chunks in the write buffer using their fragmentation metric. CBR aims to guarantee the deduplication ratio to exceed a limit with the cost of deduplicating some fragmented chunks. SMR mainly aims to gain better restore performance. Capping and NED share the similar design goal with SMR, but they overlook the redundancy among containers. Hence, redundant chunks in different containers are all counted in the utilization for each container. But one chunk can deduplicate and restore all chunks with identical contexts, and other redundant chunk copies are unreferenced. SMR considers the redundancy among containers and eliminates redundant copies counted in the utilization for containers. As shown in Section IV-C, compared with Capping and NED, SMR achieves better restore performance and higher deduplication ratio. Moreover, HAR [9] classifies fragmentation into two categories: sparse and out-of-order containers. It exploits historical information of backup versions to identify sparse containers and rewrites chunks in sparse containers to improve data locality. In fact, HAR is orthogonal with SMR. HAR rewrites the chunks in the sparse containers which can offer less referenced chunks for multiple consecutive backups. SMR rewrites chunks in each backup phase mainly to ensure high restore performance for the current backup. Chunks which are not rewritten in HAR are further examined by SMR.

## VI. CONCLUSION

The fragmentation problem significantly decreases the restore performance in chunk-based deduplication systems. We observe that existing rewriting schemes addressing the fragmentation problem often result in significant redundancy among containers, decreasing the deduplication ratio and causing redundant chunks to be read from disks to restore the backup, which wastes limited disk bandwidth and decrease the restore performance. The main challenge to alleviate the fragmentation is how to select suitable referenced containers to perform deduplication during the backup. In order to address this problem, this paper proposes a submodular maximization rewriting scheme (SMR). SMR formulates this challenge as an optimal container selection problem, which is addressed by building a submodular maximization model. The salient feature of SMR is to reduce the number of redundant and

unreferenced chunks in selected containers, alleviating the waste of disk accesses caused by unreferenced and redundant chunks to improve the restore performance. Our experimental results based on two real-world datasets demonstrate SMR outperforms the state-of-the-art work in terms of both restore performance and deduplication ratio.

## ACKNOWLEDGEMENT

This work was supported by National Key Research and Development Program of China under Grant 2016YFB1000202.

## REFERENCES

- [1] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage." in *Proc. FAST*, 2009.
- [2] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Proc. FAST*, 2008.
- [3] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace, "Metadata considered harmful... to deduplication." in *Proc. HotStorage*, 2015.
- [4] J. Ma, R. J. Stones, Y. Ma, J. Wang, J. Ren, G. Wang, and X. Liu, "Lazy exact deduplication," in *Proc. MSST*. IEEE, 2016.
- [5] Z. Sun, G. Kuenning, S. Mandal, P. Shilane, V. Tarasov, N. Xiao, and E. Zadok, "A long-term user-centric analysis of deduplication patterns," in *Proc. MSST*. IEEE, 2016.
- [6] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: coarse-grained data reordering to improve compressibility." in *Proc. FAST*, 2014, pp. 257–271.
- [7] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. SIGOPS*. ACM, 2001.
- [8] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *Proc. FAST*, 2002.
- [9] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *Proc. USENIX ATC*, 2014.
- [10] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, 2016.
- [11] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. FAST*, 2013.
- [12] R. Lai, Y. Hua, D. Feng, W. Xia, M. Fu, and Y. Yang, "A near-exact defragmentation scheme to improve restore performance for cloud backup systems," in *Proc. ICA3PP*. Springer, 2014.
- [13] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality." in *Proc. FAST*, 2009.
- [14] W. Preston, "Restoring deduped data in deduplication systems," *Search-DataBackup.com*, 2010.
- [15] Y. Nam, G. Lu, N. Park, W. Xia, and D. H. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *Proc. HPCC*. IEEE, 2011.
- [16] Y. J. Nam, D. Park, and D. H. Du, "Assuring demanded read performance of data deduplication storage with backup datasets," in *Proc. MASCOTS*. IEEE, 2012.
- [17] W. C. Preston, "Backup & recovery," 2006.
- [18] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "idedup: latency-aware, inline data deduplication for primary storage." in *Proc. FAST*, 2012.
- [19] "Gcc, the gnu compiler collection," <https://gcc.gnu.org/>.
- [20] "Linux kernel." <http://www.kernel.org/>.
- [21] A. Krause and D. Golovin, "Submodular function maximization," *Tractability: Practical Approaches to Hard Problems*, vol. 3, no. 19, p. 8, 2012.
- [22] S. Tschitschek, R. K. Iyer, H. Wei, and J. A. Bilmes, "Learning mixtures of submodular functions for image collection summarization," in *Proc. NIPS*, 2014.

- [23] P. Sinha, S. Mehrotra, and R. Jain, "Summarization of personal photographs using multidimensional content and context," in *Proc. ICMR*. ACM, 2011.
- [24] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher, "An analysis of approximations for maximizing submodular set functions," *Mathematical Programming*, 1978.
- [25] J. Edmonds, "Submodular functions, matroids, and certain polyhedra," *Combinatorial structures and their applications*, 1970.
- [26] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication performance in backup workloads," in *Proc. FAST*, 2015.
- [27] J. Liu, Y. Chai, X. Qin, and Y. Xiao, "PLC-cache: Endurable SSD cache for deduplication-based primary storage," in *Proc. MSST*. IEEE, 2014.
- [28] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proc. INFOCOM*. IEEE, 2015, pp. 1337–1345.
- [29] W. Xia, H. Jiang, D. Feng, and L. Tian, "Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets," in *Proc. DCC*. IEEE, 2014, pp. 203–212.
- [30] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Proc. MASCOFS*. IEEE, 2009, pp. 1–9.
- [31] E. Kruus, C. Ungureanu, and C. Dubnicki, "Bimodal content defined chunking for backup streams," in *Proc. FAST*, 2010, pp. 239–252.
- [32] M. O. Rabin *et al.*, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.
- [33] B. Mao, H. Jiang, S. Wu, and L. Tian, "Pod: Performance oriented i/o deduplication for primary storage systems in the cloud," in *Proc. IPDPS*. IEEE, 2014.
- [34] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *Proc. SYSTOR*. ACM, 2012.