FGDEFRAG: A Fine-Grained Defragmentation Approach to Improve Restore Performance

Yujuan Tan*, Jian Wen*, Zhichao Yan†, Hong Jiang†, Witawas Srisa-an‡, Baiping Wang*, Hao Luo§

*College of Computer Science, Chongqing University, China
Email: tanyujuan@gmail.com, b615350236@gmail.com, wbpbox@live.com

†University of Texas Arlington, Email: yanzhichao.hust@gmail.com, hong.jiang@uta.edu

‡University of Nebraska Lincoln, Email:witty@cse.unl.edu

§Nimble Storage, Email:hluo@cse.unl.edu

Abstract

In deduplication-based backup systems, the removal of redundant data transforms the otherwise logically adjacent data chunks into physically scattered chunks on the disks. This, in effect, changes the retrieval operations from sequential to random and significantly degrades the performance of restoring data. These scattered chunks are called *fragmented data* and many techniques have been proposed to identify and sequentially rewrite such fragmented data to new address areas, trading off the increased storage space for reduced number of random reads (disk seeks) to improve the restore performance. However, existing solutions for backup workloads share a common assumption that every read operation involves a large fixed-size window of contiguous chunks, which restricts the fragment identification to a fixed-size read window. This can lead to inaccurate detections due to false positives since the data fragments can vary in size and appear in any different and unpredictable address locations.

Based on these observations, we propose FGDEFRAG, a Fine-Grained defragmentation approach that uses variable-sized and adaptively located data groups, instead of using fixed-size read windows, to accurately identify and effectively remove fragmented data. When we compare its performance to those of existing solutions, FGDEFRAG not only reduces the amount of rewritten data but also significantly improves the restore performance. Our experimental results show that FGDEFRAG can improve the restore performance by 19% to 262%, while simultaneously reducing the rewritten data by 29% to 70%.

I. Introduction

Data deduplication is a lossless compression technology that has been widely used in backup [19], [8], [1], [17] and archival systems [12], [18], [2]. It breaks data streams into approximately equal-sized data chunks that are each uniquely "fingerprinted" [13] to identify chunk-level data redundancy. A chunk fingerprint is generated by a secure hash algorithm [11] according to the content in that chunk. If two fingerprints of two chunks generated by applying the same hash algorithm are identical, they are regarded as duplicate, or redundant chunks and only one instance is stored; the other chunk is then replaced by an address pointer to the stored instance. In backup systems, the incrementally changing nature of the data streams leads to a very high compression ratio, typically from 10x to 100x since a large percentage of the data is redundant among different backup versions.

While data deduplication significantly increases storage space efficiency, it also substantially complicates the post-deduplication storage management [14], [4]. For example, due to the removal of redundant chunks, the logically adjacent data chunks that belong to a specific file or data stream are scattered in different places on disks, transforming the retrieval operations of such files or data streams from sequential to random. This significantly increases the retrieval time because of the extra disk seeks, with the worst case of one seek per chunk.

Fig. 1 illustrates this problem with a simple but intuitive example. In Fig. 1, File *A* and file *A'* share the common chunk *C*. When file *A'* enters the backup system after file *A*, only chunks *E* and *F* of file *A'* would be stored since chunk *C* is already stored by file *A*, and thus chunk *C* is stored separately (non-sequentially) from chunks *E* and *F*. Thus reading file *A'* requires at least two disk seeks, one for chunk *C* and another for chunks *E* and *F*. Generally, we call a chunk such as chunk *C* as fragmented data of file *A'*. If chunk *C* is not large enough to amortize the extra disk seek overhead, this fragmentation problem can result in excessive disk seeks and lead to poor restore performance that can degrade the recovery time objective (RTO). RTO is a very important performance metric for any customers

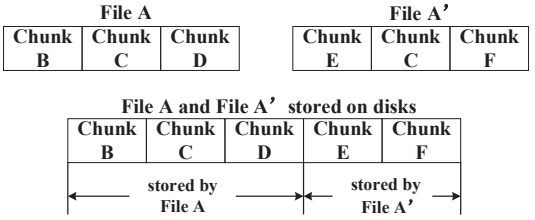


Fig. 1: An example of fragmented data.

who buy backup products.

The fragmentation problem in deduplication-based backup systems has been identified and studied to a certain extent by both industry and academia [3], [7], [5], [9], [10], [6]. To restore a backup stream, all these existing approaches have made a common, fundamental assumption that each read operation involves a large fixed number of contiguous chunks with a single disk seek. With this assumption, the disk seek time is sufficiently amortized to become negligible for each data read operation, and the read performance is determined by the percentage of referenced chunks per read in each backup stream. That is, existing approaches identify fragmented data based on the percentage of the referenced chunks in each fixed-size window (i.e., the size of the read data). If the percentage of the referenced chunks is smaller than a preset threshold, these referenced chunks will be identified as fragmented data and rewritten to be with the unique chunks of the same data stream to make future reads more sequential, trading off the increased storage space for a reduced number of reads (disk seeks) in the restore process. Taking Fig. 1 for example, chunk *C* may be rewritten to be with chunks *E* and *F* for File *A'* to read them more sequentially.

Unfortunately, while the existing approaches can improve the data restore/read performance by identifying and rewriting the fragmental chunks, they fail to accurately identify and effectively remove data fragmentation. More specifically, since the amount of data involved in each read in the existing approaches is assumed to be a fixed-size unit, the identification of fragmented data is restricted within a fixed-size window (i.e., the size of the read data). But in reality, fragmented data vary in size and can appear in different, unpredictable address locations. Detecting fragmented data in a fixed-size window can restrict the size and location of the fragmented data that can be identified and cause many false positive detections.

Consider, for example, a group of referenced chunks stored sufficiently close to one another that they either reside in a single read window but fail to meet the preset percentage threshold of referenced chunks, or meet the threshold but are split into two neighboring read windows where the split parts in each neighborhood fail to meet the threshold. Clearly, in both cases this group of closely stored reference chunks would be identified as fragmented data when they actually are not, thus resulting in false positive identification. These false positive detections can lead to rewriting more fragmental chunks but without substantially improving the restore performance.

Based on these aforementioned analysis and observations, we propose FGDEFRAG, a Fine-Grained defragmentation approach to improve restore performance in deduplication-based backup systems. The main idea and salient feature of FGDEFRAG is to *use variable-sized and adaptively located groups, instead of the fixed-size windows in the existing approaches, to identify fragmented data and atomically read data for data restores*. Specifically, FGDEFRAG first divides the data stream into variable-sized logical groups based on the on-disk store address affinity of the referenced chunks, i.e., contiguous and/or close-by referenced chunks are grouped into the same group and far-apart chunks are separated into different groups. Then for each logical group, it identifies fragmented data by comparing the *valid read bandwidth*, defined to be the total data volume of the referenced chunks of this group divided by the time spent reading the entire group (referenced and non-referenced chunks) and doing disk seek, to a preset bandwidth threshold. If it is smaller than the threshold, the corresponding referenced chunks will be identified as fragmental chunks. Finally, FGDEFRAG organizes the fragmental chunks and the new unique chunks of the same backup stream into variable sized physical groups and writes them to disks in batches. In data restores, the variable sized physical and logical groups are each read atomically.

When we compare its performance to those of the existing approaches, FGDEFRAG has two unique advantages. First, FGDEFRAG identifies the fragmented data based on the variable-sized groups and the measure of valid read bandwidth of the referenced chunk for each group, enabling it to accurately identify fragmental chunks and only rewrite a minimal number of redundant data chunks with increased spatial locality to improve the restore performance. Second, FGDEFRAG reads variable-sized groups based on the address affinity of referenced chunks, rather than reading a fixed large number of chunks each time regardless of the disk addresses of the referenced and non-referenced chunks, enabling it to accurately locate and read the referenced chunks with fewer disk seeks and a smaller amount of data to improve the data restore performance. Our experimental results show that FGDEFRAG can achieve a restore performance improvement between 19% to 262% while simultaneously reducing the amount of

rewritten data by 29% to 70%, when compared to the existing defragmentation approaches.

The rest of this paper is organized as follows. Section II describes the related work and our observations to motivate the FGDEFRAG. The design and implementation are detailed in Section III. Section IV evaluates FGDEFRAG and Section V concludes the paper.

II. Background and Motivation

For a given backup stream, fragmented data that are stored separately from its unique chunks, requires many more disk seeks to restore them than if they were stored sequentially with the unique chunks. In this section, we first review the state-of-the-art defragmentation approaches to help understand how they identify and remove the fragmental redundant chunks, and then analyze their common problems and present our observations to motivate FGDEFRAG approach.

A. Related Work on Defragmentation

In general, all existing defragmentation solutions for backup workloads define and quantify fragmented data based on the percentage of the data that belongs to a given backup stream contained in a fixed-size atomic read operation. We use an example shown in Fig. 2 to help illustrate the main ideas of the existing approaches.

In Fig. 2, there are two data objects, data object 1 with 20 chunks and data object 2 with 13 chunks. Data object 1 and data object 2 share 7 common chunks, *B*, *C*, *H*, *I*, *J*, *O*, and *Q*. All the chunks are stored in fixed-size containers of five chunks each on disks. For data *object 1*, all of its chunks are stored sequentially in the first 4 containers. But for data *object 2*, without applying any defragmentation approaches, its unique chunks are stored sequentially in the next two containers, 5 and 6, while its redundant (duplicate) chunks are stored separately via address pointers among the first 4 containers initiated by data *object 1*, as shown in Fig. 2(a). In this case, while no duplicate chunks are stored, it would require reading all 6 containers (1-6, 6 disk accesses) to restore data *object 2*, assuming that a container is the atomic read unit. When applying any of the defragmentation approaches, however, some of the redundant chunks is identified as fragmented data and rewritten to containers 5 and 6, to be stored along with the unique chunks of data *object 2*, as shown in Figures 2(b), 2(c), and 2(d). Next we describe some of the state-of-the-art defragmentation approaches.

1) HAR: The history rewriting algorithm (HAR) [3] uses a 4MB container as the atomic unit for data read operations and classifies the fragmented data into two categories based on container types, out-of-order

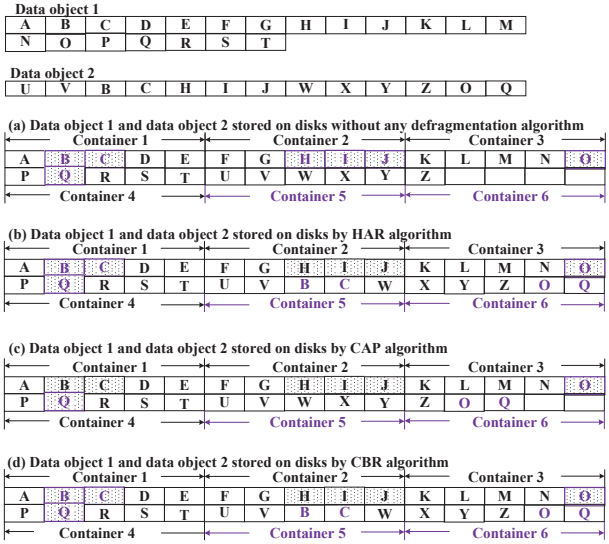


Fig. 2: An example of different data layouts on disks in existing solutions for backup workloads. Note that gray-out chunks indicate the redundant chunks.

containers and sparse containers. An out-of-order container is accessed intermittently and frequently during a restore, leading to degraded data read performance because of repeated disk accesses in a short period of time. This out-of-order container-induced problem is amenable to a cache-based solution because of the high temporal access locality of this type of containers and thus can be solved by employing powerful cache replacement algorithms, such as Assembly Area used in CAP [7], OPT used in HAR [3], and LFK [6]. A sparse container is one for which the percentage of the referenced chunks is smaller than a preset threshold that indicates an insufficient amount of valid data in each read of the container. For example, in Fig. 2, if data *object 2* represents a backup stream and the threshold is set to be 50%, the three containers, 1, 3 and 4, are regarded as sparse containers. HAR rewrites the referenced chunks in these sparse containers, *B*, *C*, *O* and *Q*, to containers 5 and 6, to be stored among the unique chunks of data *object 2*, as shown in Fig. 2(b). Thus, with HAR, one only needs to read three containers, 2, 5 and 6, to restore data *object 2*, at the cost of storing 4 duplicate chunks *B*, *C*, *O* and *Q*.

2) CAP: The capping algorithm (CAP) [7] also assumes that each atomic read involves a fixed-size container and identifies fragmented data according to the number of containers that are referenced by a fixed-size segment in a backup stream, where a fixed-size segment is a small part in a backup stream that is composed of a fixed number of contiguous chunks. For a segment, if the number of the referenced containers is larger than a preset integer M, CAP would select the top

M containers that contain the most referenced chunks as non-fragmental containers, and correspondingly, the remaining containers that contain fewer referenced chunks are then identified as fragmental containers and their referenced chunks are organized to be rewritten to new containers sequentially with the unique chunks. Again, taking Fig. 2 as an example, if data *object 2* represents a data segment and the integer *M* is 2, CAP identifies the two referenced containers, 3 and 4 that have only one referenced chunk each, as fragmental containers. The referenced chunks in these two fragmental containers, *O* and *Q*, are then rewritten to containers 5 and 6, as shown in Fig. 2(c). Note that, with CAP, four containers, 1, 2, 5 and 6, need to be read to restore data *object 2*, but at a lower redundancy cost than HAR, storing only two duplicate chunks, *O* and *Q*.

3) CBR: The context-based rewriting algorithm (CBR) [5] uses a measure called Rewrite Utility to decide whether a given referenced chunk is fragmented data, which is different from CAP and HAR that identify an entire group of referenced chunks as fragmented data. Rewrite Utility is defined to be the size of the chunks that are in the disk context but not in the stream context divided by the size of the chunks in the disk context. Disk context, in this case, is defined as a set of chunks following the decision chunk on disk, and stream context is defined as a set of chunks following the decision chunk in the backup stream, where the decision chunk is defined to be a chunk that will be identified as a fragmental chunks or non-fragmental chunk in the near future. Both of the stream context and disk context are of fixed sizes, and the size of the disk context is just the size of the data volume read in each atomic read operation, always setting to 2MB empirically. For a decision chunk, if the Rewrite Utility is higher than a preset minimal value, this chunk will be regarded as a fragmental chunk. For the example in Fig. 2, if the stream context is the 10 chunks following the decision chunk and the disk context is five chunks following the decision chunk, and the minimal value is set to be 75%, the four referenced chunks by data *object 2*, *B*, *C*, *O* and *Q*, would be regarded as fragmental chunks, since their Rewrite Utility are 80% or 100%, higher than 75%. These fragmental chunks would be rewritten to containers 5 and 6 along with the unique chunks of data *object 2*, shown in Fig. 2(d).

4) Other approaches: HAR, CAP and CBR, are three prominent defragmentation solutions for deduplication-based backup systems. In addition, *iDedup*[15] is a dynamic defragmentation solution for primary storage workloads. Nam et al. [9] use a quantitative metric to measure the fragmentation level and propose a selective deduplication scheme[10] to reduce the data fragmentation for backup workloads. The metric is calculated based on how many containers that are referenced by a backup stream, which is similar to CAP.

B. Motivation

Our review of existing defragmentation solutions for backup workloads reveals a common, fundamental assumption they share; that is, each read operation in the restore process involves a large fixed number of contiguous chunks. The main rationale for this assumption is to amortize the disk seek time with a long data transfer time in the read operation. However, when restoring a backup stream, the effective data restore performance is determined not only by the total amount of time but also by the percentage of the referenced chunks per read. Thus, existing approaches identify the fragmented data based on the percentage of the referenced chunks in each fixed-size window (i.e., the size of the read data) and rewrite the fragmented data to be among the unique chunks of the same data stream to improve data restore performance, trading off increased storage space for reduced number of reads (disk seeks) in the restore process. Ideally, the additional storage space for storing the duplicate chunks identified as fragmented data should be as small as possible while the number of disk seeks is kept at a minimum. This clearly requires the detection of fragmented data to be highly accurate and efficient. Unfortunately, fragments detection based on using fixed-size read windows in existing defragmentation approaches for backup workloads could be very inaccurate.

For a given backup stream, the referenced chunks can be grouped into different data regions naturally based on their on-disk store address proximity and affinity, i.e., contiguous or closely-located referenced chunks are grouped into the same region and far-apart chunks are separated into different regions, resulting in regions of different sizes. Taking Fig. 2 for example, chunks *B* and *C* can be grouped into the same region with a size of two chunks, while chunks *G*, *H*, and *I* can be grouped together into a region with a size of three chunks. Thus, for a given data region, the disk seek time accounts for a different percentage of the total reading time depending on the size of the region, making the disk-seek overhead different for a differently-sized region. That is, a larger region would result in a lower the disk-seek overhead, and vice versa. Therefore, treating all redundant chunks equally by grouping them into fixed-size window to identify the fragmental chunks and atomically read data for data restores, like fixed-size containers in HAR and CAP and fixed-size disk context in CBR, would surely result in a mis-opportunity to explore and exploit the address affinity of these redundant chunks of the data

stream to optimize the restore/read performance. These observations motivate us to propose FGDEFRAG to use variable-sized and adaptively located data regions based on address affinity, instead of the fixed-size regions of the existing approaches, for both identifying and removing fragmented data and atomically reading data during data restores, to optimize the restore performance in deduplication-based backup systems.

III. FGDEFRAG Approach

In this section, we first present FGDEFRAG's architecture and then describe the design of its key functional components.

A. Architectural Overview

FGDEFRAG is composed of three key functional components: data grouping, fragment identification, and group store. Fig. 3 shows its architecture and critical data path. Data grouping divides the referenced redundant chunks of each backup stream into variable-sized logical groups according to their on-disk addresses affinity, where the redundant chunks and their on-disk addresses are found by inquiring the fingerprint index table. After logical groups are identified and generated, the fragment identification component examines each group to determine whether its references to redundant chunks are fragmented data by measuring the valid read bandwidth. If the valid read bandwidth is smaller than a preset threshold, the corresponding referenced chunks are identified as fragmental chunks. Finally, the group store component organizes the fragmental chunks and the unique chunks of each backup stream into variable sized physical groups and writes them to the group pool on the disk, where a group table is used to store the start and end addresses of each physical group for future group retrieval. To restore a backup stream, FGDEFRAG uses a group cache that is able to integrate any appropriate cache replacement algorithm to improve the restore speed. Next, we describe these three components and illustrate how FGDEFRAG identifies and removes the fragmented data.

B. Data Grouping

Data grouping focuses on the referenced redundant chunks within each backup stream. Because the fragmented data is caused by the redundant chunks being stored separately from the unique chunks, it requires multiple extra disk seeks for restoration. We next describe the grouping process for the redundant chunks and show a critical factor, called *group gap*, that affects the data grouping efficiency.

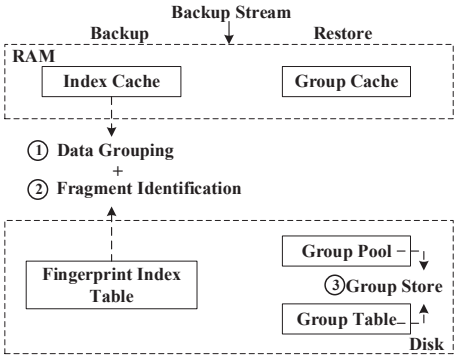


Fig. 3: FGDEFRAG's Architecture.

1) The Grouping Process: For each backup stream, data grouping is carried out in two steps.

In the first step, the backup stream is divided into large fixed-size segments [7] and the redundant chunks in each segment is sorted according to their on-disk addresses. In our current design, each segment is set to be 16MB by default. The redundant chunks and their on-disk addresses are identified by searching the fingerprint of each chunk in the fingerprint index table. A hit in the table means that the decision chunk is a redundant chunk and an identical chunk has already been stored on the disks; otherwise, the chunk is unique.

In the second step, the sorted redundant chunks in each segment are divided into variable-sized logical groups according to their address affinity, i.e., contiguous and/or closely-located chunks are grouped into the same group and far-apart chunks are separated into different groups, where each logical group starts and ends with a redundant chunk. Note that, if the redundant chunks in one logical group are not contiguous, it can contain some non-referenced chunks (i.e., chunks not belonging to the segment being processed) in the address space.

Fig. 4 illustrates the data grouping process for a segment. Fig. 4(a) shows the original sequence of the redundant chunks in the segment and Fig. 4(b) shows these redundant chunks sorted in ascending order of their addresses, the result of Step 1. After Step 2 of the process, Fig. 4(c) shows the three logical groups formed according to their address affinity.

2) The Group Gap: A critical issue for data grouping is how to determine quantitatively whether two redundant chunks in a certain proximity are either close enough to be in the same logical group or sufficiently far apart to be separated into two different logical groups. If two chunks are far away from each other, i.e., they are separated by many non-referenced chunks in an address space, but are placed in the same logical group, the relatively high percentage of non-referenced chunks in the group would significantly affect the fragment identification in a negative way. Taking the case in Fig.

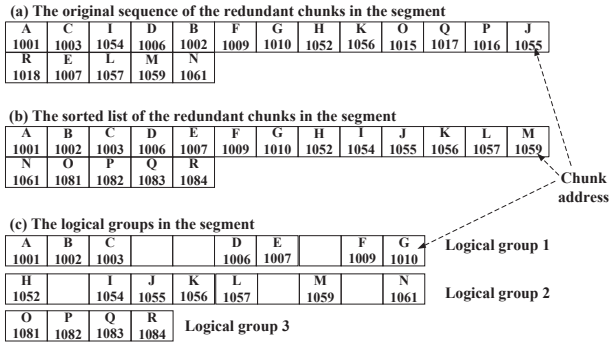


Fig. 4: An example of the data grouping process.

4(c) as an example, if chunk *H* in group 2 were instead included in group 1, 42 non-referenced chunks between chunk *G* (address 1010) and chunk *H* (address 1052) would be added to group 1, causing group 1 to be identified as a fragmental group with a much higher probability than when chunk *H* is in group 2, since the referenced redundant chunks would now account for a much smaller percentage of the chunks in the new group 1.

To address this issue, FGDEFRAG uses *group gap*, which is a distance threshold measured in MBs, to quantitatively determine whether two referenced redundant chunks are close enough or sufficiently far away from each other. Group gap is defined to be the disk bandwidth multiplied by the disk seek time. In other words, *two redundant chunks are considered sufficiently far apart to be placed in two different logical groups if the amount of non-referenced data between them takes the disk a time equal to or greater than its disk seek time to transfer (read or write), and they are considered close enough to be in the same logical group otherwise*. The rationale for using this distance threshold is based on the fact that each group is read atomically with one disk seek for data restoration and, if the non-referenced data between two referenced chunks needs more time to transfer than its disk seek, it is more efficient to put these two referenced chunks into two different groups in terms of valid read bandwidth of the segment.

C. Fragment Identification

For each logical group, FGDEFRAG uses the following formula to decide whether it is a fragmental group. Based on the formula, if the inequality is false then the group is considered fragmental.

$$\frac{x}{t+\frac{x+y}{B}} \geq B \cdot \frac{1}{N} \qquad (1)$$

In this inequality expression, B is the disk bandwidth, t the disk seek time, N a non-zero positive integer representing the bandwidth threshold factor explained next, x the total size of the referenced chunks in the group, and y the total size of the non-referenced chunks in the group (i.e., $x+y$ is the total size of the logical group). Note that the denominator, $t+\frac{x+y}{B}$, on the left hand side of the inequality represents the total time required to read the entire logical group, including the disk seek time, and the whole expression, $\frac{x}{t+\frac{x+y}{B}}$, represents the effective bandwidth of reading all the referenced data during this time, or the *valid read bandwidth* as mentioned earlier. The expression on the right hand side of the inequality, $B \cdot \frac{1}{N}$, represents the *bandwidth threshold*, a given fraction of the full disk bandwidth B. In other words, *a logical group is considered a* fragmental group *and its referenced chunks regarded as* fragmental chunks *if the valid read bandwidth is smaller than the bandwidth threshold.* This formula enables FGDEFRAG to accurately identify the fragmented data because it strictly follows the data reading process, including the disk seek and data reading, to calculate the valid read bandwidth of the referenced chunks.

D. Group Store

FGDEFRAG stores two kinds of groups, *logical groups* and *physical groups*, in order to read the referenced chunks of each segment for data restoration.

1) Physical Group Store: After fragmented data has been identified, FGDEFRAG organizes the fragmental chunks and the unique chunks of each segment into a single group, called a *physical group*, and then appends it to the disk. The size of such a physical group can be variable from one segment to another. To maximize the data write performance, FGDEFRAG writes multiple physical groups to disks in a batch mode. Meanwhile, it uses a *group table* to store the group information for future group retrieval, where each entry consists of three fields, group number, starting address and ending address. The group number is used to identify each group, whereas the starting address and ending address are used to locate the store addresses of each group on disks.

2) Logical Group Store: Besides the physical groups, the logical groups that are not identified as fragmental groups based on Formula (1) must also be remembered, in order for data restores to easily locate the referenced non-fragmental chunks for each segment. Similar to the case of physical groups, FGDEFRAG stores the group information of each logical group in the group table, consisting of the group number, starting address and ending address. Both the physical groups and logical groups are numbered automatically and incrementally, adding 1 for each new group from the latest group number.

However, in order to provide flexibility and efficiency, not all of the logical groups that are not identified

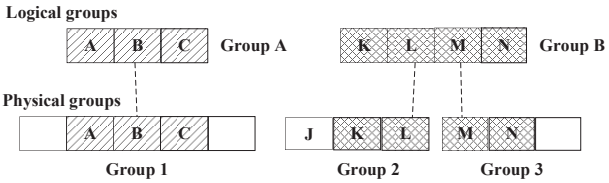


Fig. 5: Chunks' relationship between logical groups and physical groups.

as fragmental groups require to be numbered and remembered as new groups in the group table. Fig. 5 shows the relationships between the referenced non-fragmental chunks and their logical groups and the corresponding physical groups, where the chunks in each logical group are either stored in the same physical group or span across two adjacent physical groups. As indicated in Fig. 5, the logical group that completely resides in a single physical group can leverage the same information as their hosting physical group for the purpose of chunks' identification and location and thus would not be remembered as a new group.

IV. Experimental Evaluation

In this section, we assess the benefits of FGDEFRAG with an extensive experimental evaluation.

A. Experimental Setup

Baseline defragmentation approaches. We implemented FGDEFRAG in Destor [4] and compare its performance with the three state-of-the-art and prominent defragmentation approaches introduced in Section II, HAR [3], CAP [7], and CBR [5], which are referred to as baseline defragmentation approaches in this evaluation. We implemented these three defragmentation approaches in Destor, and the thresholds used for fragments identification are set to their default values based on the original publications. Specifically, the rewrite utility in CBR is set to be 0.7, the percentage threshold used to identify sparse containers in HAR is set to be 0.5, and the M used to identify fragmental containers in CAP is set to be 8 with a container size of 4MB and a segment size of 16MB. In FGDEFRAG, we set the bandwidth threshold factor N to be 10. For evaluating the restore performance, we implement two other baseline non-defragmentation approaches that do not use any rewrite algorithm but use the LRU and OPT cache algorithms for restore optimizations.

Performance metrics and evaluation objectives. We compare these approaches in two performance metrics, deduplication ratio and restore performance. Deduplication ratio, defined as the amount of data removed by deduplication divided by the total amount of data in the backup stream, is an important performance metric for defragmentation space efficiency. Since all the defragmentation approaches try to decrease data fragmentation by rewriting some fragmental chunks, at the cost of storing duplicate chunks, deduplication ratio quantifies this defragmentation space cost. In other words, a higher deduplication ratio indicates a lower space cost. Restore performance is an important performance metric for defragmentation effectiveness, since the goal of defragmentation is to improve the restore performance. For each backup stream, the restore performance is defined to be the total amount of data in the whole backup stream divided by the time spent reading the data regions containing the referenced chunks, including the time spent on data reads and disk seeks. In our experiments, each disk seek takes 10ms and the disk bandwidth is 100MB/s.

Dataset name	MAC snaphsots	fslhome
#of versions	100	11
Total size	6.36TB	3.4TB
Unique size	6.29GB	400GB

TABLE I: Characteristics of datasets.

Evaluation workloads: the datasets. The datasets used in our experiments come from the public archive traces and snapshots [16], including MAC snapshots and Fslhome dataset. The MAC snapshots dataset was collected on a Mac OS X Snow Leopard server running in an academic computer lab, whereas the Fslhome dataset contains snapshots of four students' home directories from a shared network file system. In both the datasets, the average chunk size is 8KB. Table I shows their other characteristics.

B. Deduplication Ratio

Table II compares the deduplication ratios and the amounts of the rewritten data between FGDEFRAG and the baseline approaches for all the 100 MAC snapshots and 14 versions of the Fslhome datasets. The results indicate that FGDEFRAG's deduplication ratio is 6.4% higher than CAP, 1.1% higher than CBR for the MAC snapshots dataset, and 6.5% higher than CAP, 1.5% higher than CBR for the Fslhome dataset. In the meantime, FGDEFRAG rewrites 70% and 29.4% less data than CAP and CBR for the MAC snapshots dataset, 70.6% and 36% less data than CAP and CBR for the Fslhome dataset. The significant reductions in the amount of rewritten data by FGDEFRAG is due to its accurate identification of the data fragments that minimizes false positive identifications of data fragments and thus, it rewrites much few redundant chunks to disks.

Moreover, FGDEFRAG underperforms HAR in deduplication ratios for both datasets, even though it also reduces false positive fragmental data as described in Section II-B. The main reason is that HAR has more

	Deduplication Ratio (percentage)		Rewritten Data (GB)	
	MAC	fslhome	MAC	fslhome
FGDEFRAG	96.4	91.5	163	112
CAP	90	85	542	380.8
CBR	95.3	90	231	175
HAR	98	93	50	88.2
None	99	93.5	0	0

TABLE II: Comparison between FGDEFRAG and the baseline defragmentation approaches (CAP, CBR, HAR) and baseline non-defragmentation approach (None) in terms of the deduplication ratio and amount of the rewritten data for all the backup versions of MAC snaphsots and Fslhome dataset.

Approaches	# of disk seeks	reading data(GB)
None+LRU	72402	303.68
None+OPT	55346	232.24
HAR	29420	123.4
CAP	22167	92.98
CBR	16006	67.13
FGDEFRAG+LRU	13228	59.26
FGDEFRAG+OPT	12790	57.43

TABLE III: Comparison between the FGDEFRAG and the baseline approaches in terms of the number of disk seeks and the total size of read data required to restore the last version of the MAC snapshots datasets with a 1GB cache.

false negatives; that is, it misses identifying some local fragmental chunks, and thus rewrites less redundant chunks to disks for restore performance optimization. More specifically, compared to all the other defragmentation approaches that identify in a segment (i.e., a small part of a backup stream) locally, HAR identifies the fragmental chunks in sparse containers in a whole backup stream globally. As a result, each container in HAR can have more referenced chunks in a global backup stream than that in a small segment. A container not identified as a sparse container that meets the percentage threshold of globally referenced chunks is likely the one that cannot meet the percentage threshold in a segment locally. Unable to identify such local sparse containers, HAR, therefore, rewrites fewer fragmental (redundant) chunks.

C. Restore Performance

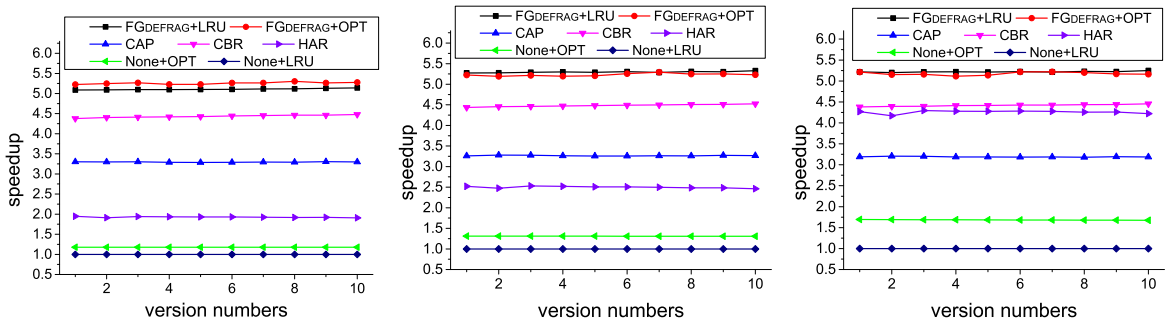
Restore performance is the most important performance metric for measuring the effectiveness of defragmentation approaches. In this subsection, we compare the restore performances between FGDEFRAG and the defragmentation baseline approaches (CAP, CBR, HAR) and the non-defragmentation baseline approach based on LRU and OPT cache algorithms (simply referred to as *None*). For CAP, CBR and HAR, various cache algorithms are employed to improve the restore performance, For example, CAP uses Assembly Area [7]; HAR uses OPT [3]; and CBR uses LFK [6]. For the FGDEFRAG and non-defragmentation baseline approaches, both LRU and OPT cache algorithms are implemented, i.e., FGDEFRAG+LRU/OPT, None+LRU/OPT. Both LRU and OPT cache replacement algorithms are implemented based on their basic reading/storage units. For example, FGDEFRAG evicts the least recently used group by LRU algorithm or evicts the group that will not be accessed for the longest time in the future by OPT algorithm when the cache is full.

Moreover, because the sequence of reading chunks during the restore is just the same as the sequence of writing them during a backup, we implemented the OPT cache algorithm with the help of the backup process in order to know the future chunks' access patterns. Specifically, during a backup process, when a chunk is processed through either redundancy elimination or data writing, it has a storage unit. We record the IDs of these storage units during backup processes, and then use these ID sequences to guide for chunks' reading during data restores and help OPT cache algorithm to know which storage unit that will not be accessed for the longest time in the future to make room for other storage units.

Fig. 6 and Fig. 7 compare the restore performance between FGDEFRAG and the baseline approaches to process MAC snapshots and Fslhome datasets, respectively. As shown, FGDEFRAG outperforms all the baseline approaches consistently in all cases. For the MAC snapshots dataset, FGDEFRAGE, on average, outperforms CAP, CBR and HAR by 60%, 20% and 176%, respectively when the cache size is 512MB; 63%, 19% and 116%, respectively when the cache size is 1GB, and 62%, 19.6% and 23% respectively when the cache size is 2GB. For the Fslhome dataset, FGDEFRAG outperforms CAP, CBR and HAR by 27%, 38% and 262% respectively with a 512MB cache; 30%, 37% and 217% with a 1GB cache; 35%, 38% and 159% with a 2GB cache; and 43%, 39%,and 76% with a 4GB cache.

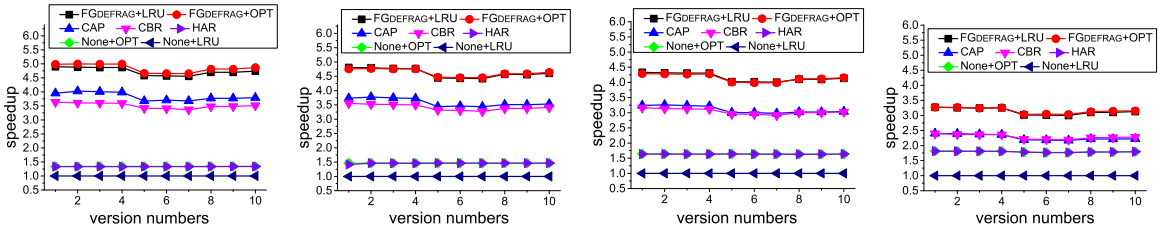
There are two reasons for such significant improvements in FGDEFRAG. First, FGDEFRAG accurately identifies the fragmental chunks, and thus only rewrites a minimal number of redundant data chunks with increased spatial locality to improve the restore performance. Second, FGDEFRAG reads variable-sized groups based on the address affinity of referenced chunks, rather than reading a fixed large amount of chunks each time regardless of the disk addresses of the referenced and non-referenced chunks. Thus, FGDEFRAG can accurately locate and read the referenced chunks with higher valid read bandwidth for data restores. The combination of these two methods enables FGDEFRAG to restore the dataset with a fewer number of disk seeks and a smaller amount of data than the baseline approaches, as detailed in Table III, leading to much higher restore performance.

Since HAR misses identifying and rewriting lo-



(a) Cache size: 512MB (b) Cache size: 1GB (c) Cache size: 2GB

Fig. 6: Comparison between FGDEFRAG and the baseline approaches in restore performance with the last 10 versions of the MAC snapshots dataset. Speedup represents the restore performance normalized by that of the None+LRU approach.



(a) Cache size: 512MB (b) Cache size: 1GB (c) Cache size: 2GB (d) Cache size: 4GB

Fig. 7: Comparison between FGDEFRAG and the baseline approaches in restore performance with the last 10 versions of the Fslhome dataset. Speedup represents the restore performance normalized by that of the None+LRU approach.

cal fragmental chunks for restore performance optimizations (as described in Section IV-B), its performance is only comparable to that of the baseline non-defragmentation approach using the OPT cache replacement algorithm when the cache size is either 512MB, 1GB, 2GB or 4*GB* when it is used to process the Fslhome dataset. For the MAC snapshot dataset, it has the lowest restore performance among the three baseline defragmentation approaches when the cache size is 512MB and 1GB respectively. The main reason is that HAR cannot capture the spatial locality of the local fragmental chunks in the restore cache, and thus it needs more accesses to disks for those fragmental chunks it failed to identify. When the cache size is increased to 2GB for the MAC snapshot dataset, HAR outperforms CAP and CBR because of the increased spatial locality captured by a large cache. However, due to the hardware cost and multi-user/multi-job environment where the limited cache space is shared among multiple applications, HAR's advantage under the assumption of a large cache is likely to either diminish or become very costly. Moreover, because the MAC snapshots and Fslhome datasets only require a disk space of 63GB and 191 GB, respectively to store their unique data chunks, we did not evaluate their restore performance under a cache size of more than 2 GB for the MAC snapshot dataset and 4 GB for Fslhome dataset.

D. Sensitivity Studies

In the design space of FGDEFRAG, there are one important design parameter, namely, the bandwidth threshold factor *N*, that significantly impacts both the deduplication ratio and restore performance metrics. FGDEFRAG identifies fragmental groups by comparing the valid read bandwidth to a specified bandwidth threshold (i.e., factor *N* in Formula(1)). Thus different bandwidth thresholds are likely to have different impacts on the deduplication ratios and restore performance measures. Fig. 8 shows FGDEFRAG's deduplication ratio and restore performance as a function of the bandwidth threshold factor *N* while using the LRU algorithm and OPT algorithm, respectively. The experiment restores the last 20 backup versions of the MAC snapshots dataset.

As seen from the results, the deduplication ratio increases with *N*, especially from 2 to 4 when it increases from 88% to 94%. On the other hand, the restore performance decreases significantly as *N* increases. For example, under the LRU algorithm with a 256MB cache, the restore performance is reduced by about 30% when *N* grows from 2 to 18. This is because when *N* increases, FGDEFRAG is likely to identify fewer logical groups as fragmental groups and fewer fragmental (redundant) chunks would be rewritten to disks to improve restore performance, but also resulting in higher deduplication ratio. To properly trade off

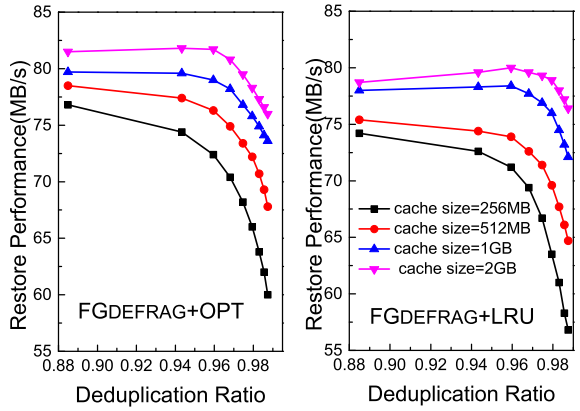


Fig. 8: Sensitivity of the restore performance and deduplication ratio to the value of the bandwidth threshold factor N. The restore performance is the average value of the last 20 versions of the MAC snapshots dataset. The 9 data points on each curve represent the corresponding restore performance and deduplication ratio for each of the 9 N values, 2, 4, 6, 8 10, 12, 14, 16, 18, from left to right.

between deduplication ratio and restore performance, we need to select appropriate values of N for different datasets. In our experimental datasets, the appropriate N values are found to range from 6 to 10.

V. Conclusion

In this paper we introduce FGDEFRAG, a new defragmentation approach that is more accurate and efficient than existing defragmentation approaches. FGDEFRAG is a fine-grained approach that uses variable-sized and adaptively located logical chunk groups based on the address affinity of the chunks, to identify and remove fragmentation. FGDEFRAG's high accuracy in fragmentation detection and effective exploitation of chunk locality enable it to improve both the restore performance and deduplication ratio. Our experimental results show that FGDEFRAG outperforms three state-of-the-art defragmentation schemes, CAP, CBR and HAR in restore performance by 27% to 63%, 19% to 39%, 23% to 262%. In terms of deduplication ratios, FGDEFRAG also outperforms CAP and CBR but slightly underperforms HAR, because HAR identifies the fragmental chunks globally but at the expense of missed detection of some local fragmental chunks; and therefore, HAR rewrites fewer redundant chunks, leading to slightly higher deduplication ratio.

Acknowledgment

The authors wish to thank the reviewers for their constructive comments. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No.61402061, No.61672116, Chongqing Basic and Frontier Research Project of China under Grant No.cstc2016jcyjA0274, No.cstc2016jcyjA0332, Research Fund for the Doctoral Program of Higher Education of China Under Grant No.20130191120031, NSF CCF-1629625 and NetApp grant.

References

[1] D. Bhagwat, K. Eshghi, D. D.E. Long, and M. Lillibridge. Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup. Technical Report HPL-2009-10R2, HP Laboratories, Sep. 2009.

[2] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. Hydrastor: A scalable secondary storage. In *FAST'09*, Feb. 2009.

[3] 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 *USENIX ATC'14*, Jun. 2014.

[4] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhuang, and Y. Tan. Reducing fragmenation impact with forward knowledge in backup systems with deduplication. In *USENIX FAST'15*, Feb. 2015.

[5] M. Kaczmarczyk, M. Barczynski, and C. Dubnicki. Reducing impact of data fragmentation caused by in-line deduplication. In *ACM SYSTOR'12*, Jun. 2012.

[6] M. Kaczmarczyk and C. Dubnicki. Reducing fragmentation impact with forward knowledge in backup systems with deduplication. In *ACM SYSTOR'15*, Jun. 2015.

[7] M. Lillibridge, K. Eshghi, and D. Bhagwat. Improving restore speed for backup systems that use inline chunk-based deduplication. In *USENIX FAST'13*, Feb. 2013.

[8] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Campbell. Sparse Indexing: Large scale, inline deduplication using sampling and locality. In *FAST'09*, Feb. 2009.

[9] Y. Nam, G. Park G. Lu, W. Xiao, and D. H. Du. Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage. In *IEEE HPCC'11*, Sep. 2011.

[10] Y. J. Nam, D. Park, and D. H. Du. Assuring demanded read performance of data deduplication storage with backup datasets. In *IEEE MASCOTS'12*, Aug. 2012.

[11] NIST. Secure Hash Standard. In *FIPS PUB 180-1*, May 1993.

[12] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *FAST'02*, Jan. 2002.

[13] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[14] P. Shilane, R. Chitloor, and U. K. Jonnala. 99 Deduplication Problems. In *USENIX Hotstorage'16*, Jun. 2016.

[15] K. Srinivasan, T. Bisson, G. Goodson, and Y. Voruganti. iDedup: Latency-aware, inline data deduplication for primary storage. In *USENIX FAST'12*, Feb. 2012.

[16] Traces and Snapshots Public Archive. http://tracer.filesystems.org/.

[17] W. Xia, H. Jiang, D. Feng, and Y. Hua. SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. In *USENIX ATC'11*, Jun. 2012.

[18] L. L. You, K. T. Pollack, and D. D. E. Long. Deep Store: An archival storage system architecture. In *ICDE'05*, Apr. 2005.

[19] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *FAST'08*, Feb. 2008.