Improving Restore Performance in Deduplication-Based Backup Systems via a Fine-Grained Defragmentation Approach

Yujuan Tan, *Member, IEEE*, Baiping Wang, Jian Wen, Zhichao Yan, Hong Jiang, *Fellow, IEEE*, and Witawas Srisa-an, *Member, IEEE*

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 identifications 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 14 to 329 percent, while simultaneously reducing the rewritten data by 25 to 87 percent.

Index Terms—Data deduplication, defragmentation, deduplication ratio, restore performance

1 Introduction

Data deduplication is a lossless compression technology that has been widely used in backup [1], [9], [19], [21] and archival systems [2], [13], [20]. It breaks data streams into approximately equal-sized data chunks that are each uniquely "fingerprinted" [14] to identify chunk-level data redundancy. A chunk fingerprint is generated by a secure hash algorithm [12] 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 [5], [15]. 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 who buy backup products.

The fragmentation problem in deduplication-based backup systems has been identified and studied to a certain extent by

- Y. Tan, B. Wang, and J. Wen were with the College of Computer Science, Chongqing University, Chongqing 400044, China. E-mail: {tanyujuan, b615350236}@gmail.com, comems@msn.com.
- Z. Yan and H. Jiang are with the University of Texas Arlington, Arlington, TX 76019. E-mail: yanzhichao.hust@gmail.com, hong.jiang@uta.edu.
- W. Srisa-an is with the University of Nebraska Lincoln, Lincoln, NE 76019. E-mail: witty@cse.unl.edu.

Manuscript received 5 Aug. 2017; revised 21 Dec. 2017; accepted 18 Mar. 2018. Date of publication 20 Apr. 2018; date of current version 7 Sept. 2018. (Corresponding author: Yujuan Tan.)
Recommended for acceptance by P. Sadayappan.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2018.2828842

1045-9219 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

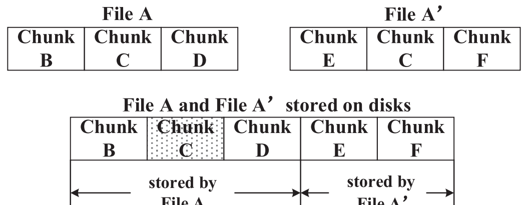


Fig. 1. An example of fragmented data.

both industry and academia [3], [4], [6], [7], [8], [10], [11], [18]. 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. Identifying 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 identifications.

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 portion in each neighborhood fails to meet the threshold. Clearly, in both cases this group of closely stored referenced chunks would be identified as fragmented data when they actually are not, thus resulting in false positive identification. These false positive identifications can lead to rewriting more fragmental chunks but without substantially improving the restore performance. Our experimental analysis, elaborated in Section 2 (Motivation) and Section 4 (Evaluation), shows that as much as 65.3 percent of the fragmented data is mistakenly identified by the state-of-the-art defragmentation approaches, which leads to rewriting up to 70 percent of extra data, but without commensurately improving the restore performance.

Besides this fixed-sized read window for fragments identifications, another problem is that all of the existing defragmentation approaches break each backup stream into fixed-size segments to find the referenced chunks and fragmental chunks. Specifically, all of the existing approaches divide each backup stream into fixed-sized windows of contiguous chunks (i.e., always called fixed-sized segments) upon receiving backup streams, no matter the backup stream is a full backup stream or incremental backup stream. If the referenced chunks in a fixed-sized segment in a backup stream are not stored closely enough on disks, those chunks would be identified as fragmental chunks. This works well for full backups since the full backups collect the whole datasets and the chunks in the backup stream and the backup dataset have the same chunk logical sequences. But for incremental backup streams, due to that each incremental backup only collects and backs up the modified data based on the last backup, the adjacent chunks emerged in a fixed-sized segment from an incremental backup stream may come from far away different files. Thus it is not correct to identify those chunks as fragmental chunks if they are not stored closely enough on disks, since those far-apart files will not be read together for data restores.

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 data regions, 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 backup stream into variable-sized segments according to the addresses affinity of the chunks in a backup dataset. If two chunks are adjacent and/or close to each other in the backup dataset, they will be put into the same segment, otherwise if they come from two far away different files, they will be separated into different segments. After segmenting the backup stream, FGDEFRAG then divides the chunks in each segment 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. 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 the following three unique advantages:

- *First*, FGDEFRAG organizes the chunks into variable-sized segments according to the chunks' addresses in the backup dataset instead of the backup stream, enabling the adjacent chunks emerged in a backup stream but actually far away from each other in the

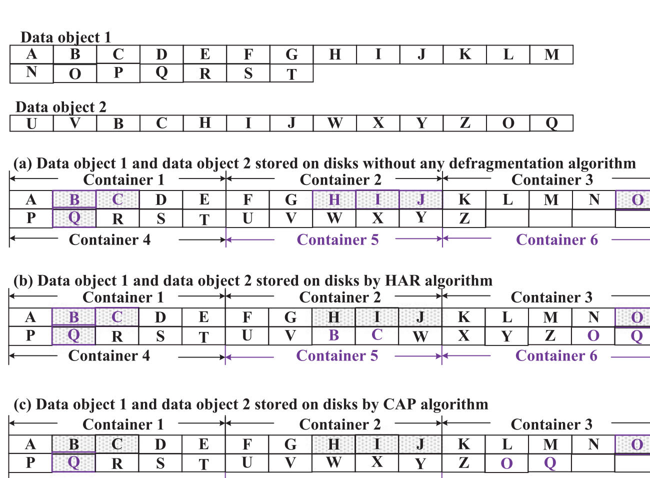


Fig. 2. An example of different data layouts on disks in existing solutions. Note that gray-out chunks indicate the redundant chunks.

backup dataset are put into different segments. This is especially useful for incremental backups since it only includes the modified data and doesn't have the correct chunk sequences as in the backup dataset.

- *Second*, FGDEFRAG identifies the fragmented data based on the variable-sized groups according to the on-disk address affinity 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.
- *Third*, 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 14 to 329 percent while simultaneously reducing the amount of rewritten data by 25 to 87 percent in overall, when compared to the existing defragmentation approaches.

The rest of this paper is organized as follows. Section 2 describes the related work and our observations to motivate the FGDEFRAG. The design and implementation are detailed in Section 3. Section 4 evaluates FGDEFRAG and Section 5 concludes the paper.

2 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.

2.1 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. 2a. 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 Figus. 2b, 2c, 2d and 2e. Next we describe some of the state-of-the-art defragmentation approaches.

2.1.1 HAR

The history rewriting algorithm (HAR) [4] uses a 4 MB container as the atomic unit for data read operations and classifies the fragmented data into two categories based on container types, out-of-order 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 [8], OPT used in HAR [4], and LFK [7]. 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 percent, 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. 2b. 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*.

2.1.2 CAP

The capping algorithm (CAP) [8] 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. 2c. 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*.

2.1.3 CBR

The context-based rewriting algorithm (CBR) [6] 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 2 MB 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 percent, 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 percent, higher than 75 percent. These fragmental chunks would be rewritten to containers 5 and 6 along with the unique chunks of data *object 2*, shown in Fig. 2d.

2.1.4 iDedup

Besides deduplication-based backup systems, a dynamic defragmentation solution, named *iDedup* [16], has been proposed in deduplication-based primary storage systems. Different from a typical backup system, a primary storage system focuses more on the latency for small data writes/reads rather than the throughput for massive data writes/reads. To reduce the fragmented data, *iDedup* removes only the duplicate sequences of chunks stored on disks. Thus, subsequent reading of those referenced chunks can be done sequentially. *iDedup* sets a minimum length of a duplicate sequence and any duplicate sequence whose length is not shorter than the minimum length is removed. For the example in Fig. 2, *iDedup* is applied to the backup workload and the minimum length is set to be 2 chunks; therefore it removes the two duplicates sequences, *B*, *C* and *H*, *I*, *J*, for data *object 2*. The other remaining duplicate chunks, *O* and *Q*, are regarded as the new chunks and written to containers 6 along with other new chunks of data *object 2*, shown in Fig. 2e.

iDedup only removes the duplicate sequences each time, without considering a group of duplicate chunks that are stored closely but are not stored continuously. Only removing such duplicate sequence would keep many out-of-sequence duplicate chunks. Moreover, a lot of short sequences exist in backup systems, especially in incremental backups. Removing such short sequences and then reading them each time would incur many disk seeks for data location, which can significantly degrade the data reading throughput since reading such small data each time cannot amortize the extra disk overhead. The evaluation section (Section 4) clearly shows that *iDedup* gains a very low restore performance for backup workloads and is not appropriate for data intensive backup systems.

2.1.5 Other Approaches

HAR, CAP and CBR, are three prominent defragmentation solutions for deduplication-based backup systems, and *iDedup* is the representing defragmentation solution for primary storage workloads. In addition, Nam et al. [10] use a quantitative metric to measure the fragmentation level and propose a selective deduplication scheme[11] 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.

2.2 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 identification of fragmented

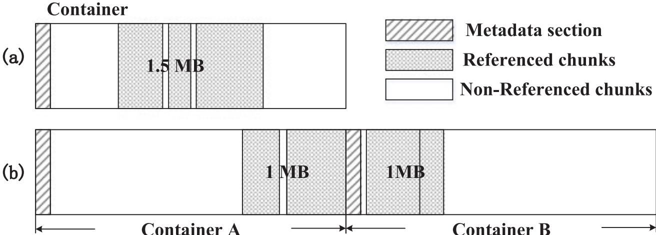


Fig. 3. The referenced chunks mistakenly identified as fragmented data in existing defragmentation solutions. Each container is 4 MB in size plus the metadata section in the header.

data to be highly accurate and efficient. Unfortunately, fragments identification based on using fixed-size read windows in existing defragmentation approaches for backup workloads could be very inaccurate as elaborated next.

2.2.1 False Positive Identification of Fragmented data

In general, there are two kinds of false positive identification of fragmented data that mistakenly identify some non-fragmented data as fragmented data in existing defragmentation approaches.

The first kind of false positives are the ones generated within a read window because of the relative size and rigidity of the fixed-size window, which can be explained by the example in Fig. 3a. In this example, a group of 1.5 MB referenced chunks that resides closely in the middle of a container. With any of the existing defragmentation approaches, say, HAR with a threshold of 50 percent, the 1.5 MB referenced chunks would be identified as fragmented data since they account for only 37.5 percent of the container, even though these closely stored 1.5 MB chunks are arguably sufficiently large to amortize the disk seek time. However, if we narrow the window of identification from the whole container to a smaller one but still large enough to contain these referenced chunks, they would no longer be identified as fragmented data since the effective read bandwidth of these referenced chunks is more than 50 percent of the disk bandwidth. Specifically, assuming that the disk bandwidth is 100 MB/s and the disk seek time is 10 ms, the effective read bandwidth of the 1.6 MB region containing these 1.5 MB referenced chunks is about 57.7 MB/s ($\frac{1.5\text{ MB}}{\frac{1.6\text{ MB}}{100\ MBs}+0.01s}$), or 57.7 percent of the disk bandwidth.

The other kind of false positives are due to a group of referenced chunks, contiguous or in close proximity, that are split by the boundary of two adjacent windows of identification as in the case of Fig. 3b. By employing any of the existing approaches, e.g., HAR with a threshold of 50 percent, the referenced chunks in both containers, which each account for 25 percent of their respective container, would be identified as fragmented data. However, if we adjust the identification window appropriately to contain both groups of the referenced chunks in a single window, these referenced chunks would not be identified as fragmented data since their effective read bandwidth is 62.5 MB/s (2.2 MB region containing 2 MB referenced chunks), or 62.5 percent of the 100 MB/s disk bandwidth and assuming a disk seek time of 10 ms.

2.2.2 Statistical Evidences of False Positives

To better understand the impact of the false positives due to inaccurate identification of fragmented data, we empirically evaluate HAR, CAP and CBR algorithms to quantify the percentages of their falsely identified fragmental chunks. The investigation is conducted in the Destor prototype system [5], where the disk bandwidth is 100 MB/s and disk seek time is 10ms, and the experimental data sets come from 100 snapshots of a Mac OS X Snow Leopard server running in an academic computer lab collected in 2011, totaling about 6.3 TB data [17]. The percentage threshold in HAR is set to be 0.5, the rewrite utility in CBR is set to be 0.7, and the integer M in CAP is set to be 8 with a container size of 4 MB and a segment size of 16 MB. All these parameters are set according to their default values based on the original publications.

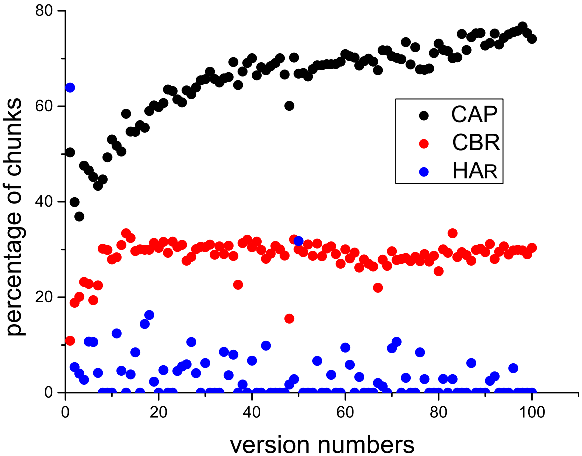


Fig. 4. Percentages of data chunks falsely identified as fragmented data by CAP, CBR and HAR.

In this investigation, a group of data chunks are considered to be falsely identified as fragmented data if these chunks are recognized as fragmented data in HAR, CBR or CAP but are not identified as fragmented data in our proposed solution by grouping them (elaborated in Section 3.2) and identifying them based on the measure of valid read bandwidth by satisfying Formula (1) (elaborated in Section 3.3). Here we set the integer N in Formula (1) to be 2. Fig. 4 shows the experimental results. As seen from Fig. 4, a large percentage of chunks are falsely identified as fragmented data by the CAP, CBR and HAR algorithms, with an average and maximum of 65.3 and 77 percent in CAP, 28.7 and 40 percent in CBR and 3.7 and 64 percent in HAR. These falsely identified chunks would then be erroneously rewritten to new addresses (next to or among the unique chunks of the same data stream) on disks to remove fragmentation, decreasing the deduplication ratio without significantly improving the restore performance. Detailed discussion about our evaluation and the reported results appear in Section 4.

2.2.3 Variable Sized Regions of Referenced Chunks

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 H, I, and J can be grouped together into a region with a size of

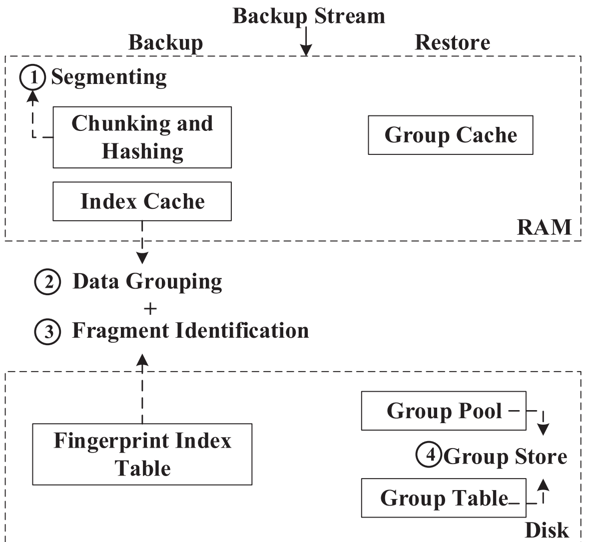


Fig. 5. FGDEFRAG's architecture.

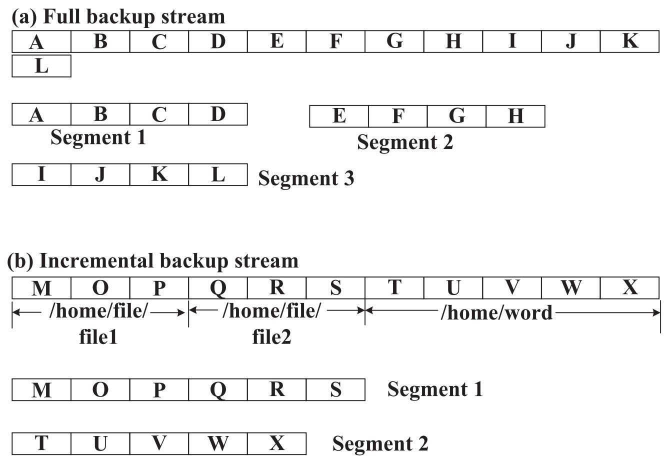


Fig. 6. Two examples of segmenting processes for both full backup stream and incremental backup stream.

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 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 missed 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.

3 FGDEFRAG APPROACH

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

3.1 Architectural Overview

FGDEFRAG is composed of four key functional components: segmenting, data grouping, fragment identification, and group store. Fig. 5 shows its architecture and critical data path. Segmenting is responsible for dividing each backup stream into fixed-sized or variable-sized data segments according to the chunks' logical addresses affinity in backup datasets. Data grouping divides the redundant chunks in each segment 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 four components and illustrate how FGDEFRAG identifies and removes the fragmented data.

3.2 Segmenting

For each backup stream, FGDEFRAG first divides the chunks into a serial of data segments by segmenting module.

FGDEFRAG uses different segmenting methods for full backup streams and incremental backup streams. For full backup streams, FGDEFRAG divides the chunks into large fixed-size data segments, like CAP [8]. Each data segment is composed of a fixed number of contiguous chunks. In our current design, the average chunk size is 8 KB and each data segment is set to be about 16 MB by default. Fig. 6a shows an example of the segmenting process for a full backup stream. In this example, the full backup stream is composed of 12 chunks. It is divided into three fixed-sized data segments and each data segment is composed of four chunks.

While for incremental backups, since it only reads the modified data based on the last backup each time, FGDEFRAG divides the chunks into variable-sized data segments according to the affinity of the chunks' logical addresses in the backup dataset. That is, if the logical addresses of two chunks are close enough, they will be put into the same data segment; otherwise, they will be separated into two different data segments. The logical addresses and the segment identification are elaborated next.

3.2.1 *The Logical Addresses*

The chunk's logical address represents the relative position of each chunk in the backup dataset. For simplistic FGDEFRAG uses the filename that the chunk belongs to to represent the approximate logical address of each chunk in the backup dataset. If two contiguous chunks are from the same file,

they would have the same logical address. FGDEFRAG uses this chunk's logical address to measure whether two contiguous chunks emerged in an incremental backup stream are close enough and can be put into the same data segment.

3.2.2 Segment Identification

In the following two situations, FGDEFRAG identifies the chunks emerged in an incremental backup stream that are close enough and can be put into the same data segment. First, if two chunks come from the same file and have the same logical addresses, they can be put into the same data segment. Second, if two contiguous chunks belong to two neighboring files residing in the same directory, that is, these two chunks' logical addresses are near to each other, FGDEFRAG will also put them into the same data segment. Except for these two situations, FGDEFRAG will identify the other contiguous chunks far away from each other and will separate them into different data segments. Moreover, in order to avoid a single segment to grow too large, FGDEFRAG sets a maximum size of 32 MB to limit the segment size.

Fig. 6b shows an example of the segmenting process for an incremental backup stream. This incremental backup stream is composed of 11 chunks that are collected from three separate files, /home/file/file1, /home file/file2 and /home/word. According to the segmenting methods for incremental backup streams, the former 6 chunks that come from two neighboring files, /home/file/file1 and /home file/file2, will be put into the same segment, and the other 5 chunks, coming from file /home/word, will be put into another data segment.

3.3 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.

3.3.1 The Grouping Process

For the chunks in each data segment from backup streams, data grouping is carried out in two steps.

In the first step, it identifies the redundant chunks in each data segment and then sorts those redundant chunks according to their on-disk addresses. 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. 7 illustrates the data grouping process for a segment. Fig. 7a shows the original sequence of the redundant chunks in the segment and Fig. 7b shows these redundant chunks sorted in ascending order of their addresses, the result of Step 1. After Step 2 of the process, Fig. 7c shows the three logical groups formed according to their address affinity.

3.3.2 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. 7c 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.

3.4 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

$$\frac{x}{t + \frac{x+y}{B}} \geq B \cdot \frac{1}{N}. \tag{1}$$

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

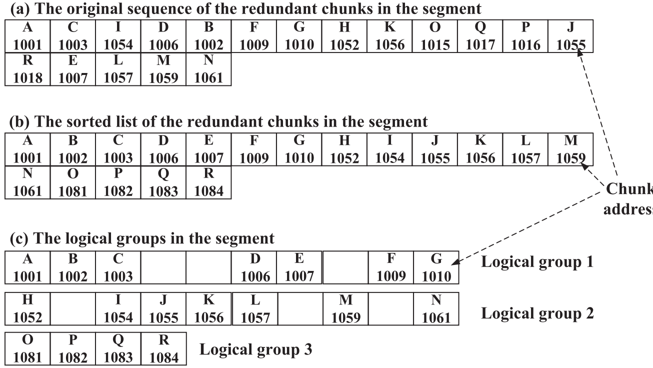


Fig. 7. An example of the data grouping process.

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.

3.5 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.

3.5.1 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.

3.5.2 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 as fragmental groups require to be numbered and remembered as new groups in the group table. Fig. 8 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. 7, 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.

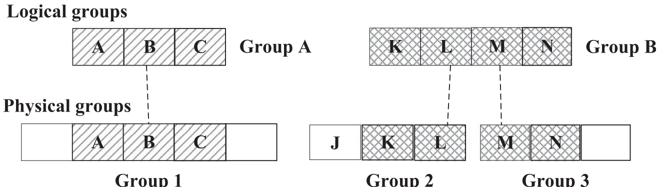


Fig. 8. Chunks' relationship between logical groups and physical groups.

4 EXPERIMENTAL EVALUATION

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

4.1 Experimental Setup

Baseline Defragmentation Approaches. We implemented FGDEFRAG in Destor [5] and compare its performance with the four state-of-the-art and prominent defragmentation approaches introduced in Section 2.1, HAR [4], CAP [8], CBR [6], and iDedup [16], which are referred to as baseline defragmentation approaches in this evaluation. We implemented these four defragmentation approaches in Destor, and the thresholds used for fragments identification are set to the optimal values to achieve the best performances with our datasets. Specifically, the rewrite utility in CBR is set to be 0.8, the percentage threshold used to identify sparse containers in HAR is set to be 0.5, the minimum length of the duplicate sequence removed in iDedup is set to be 4 chunks, and the M used to identify fragmental containers in CAP is set to be 8 with a container size of 4 MB and a segment size of 16 MB. In FGDEFRAG, we set the bandwidth threshold factor N to be 10. Besides the four baseline defragmentation approaches, we also implement two other baseline non-defragmentation approaches that do not use any rewrite algorithm. One is doing the exact deduplication (referred to as Exact) that removes all of the redundant data, while the other keeps some redundant data to accelerate the deduplication speed. For the latter one, we select Sparse [9] as a representative. To evaluate the restore performance, these two approaches use both the LRU and OPT cache algorithm to improve their restore performances.

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

TABLE 1
Characteristics of Datasets

Dataset name	MAC Snapshots		Fslhome
	Full	Inc.	
#of versions	100	33	14
Total size	6.36TB	0.34TB	3.44TB
Unique size	63GB	45GB	191GB

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 10 ms and the disk bandwidth is 100 MB/s.

Evaluation Workloads: The Datasets. The datasets used in our experiments come from the public archive traces and snapshots [17], 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. However, since the two datasets directly downloaded from the public traces only contain full snapshots, we further synthesized 32 incremental snapshots by extracting the incremental file-level data changes of 32 snapshots based on their previous ones from MAC snapshot datasets, which are used for the evaluation of incremental backups. In all of the datasets, the average chunk size is 8 KB. Table 1 shows their other characteristics.

4.2 Deduplication Ratio

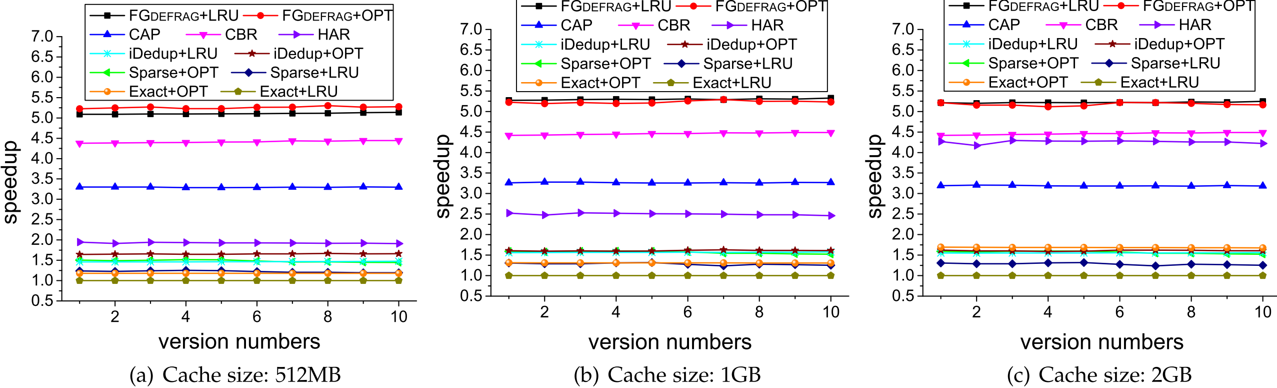
Table 2 compares the deduplication ratios and the amounts of the rewritten data between FGDEFRAG and the baseline approaches for all the 100 MAC full snapshots, 32 MAC incremental snapshots, and 14 versions of the Fslhome datasets. The results indicate that FGDEFRAG's deduplication ratio is 6.4 percent higher than CAP, 0.9 percent higher than CBR, 0.9 percent higher than iDedup for the MAC full snapshots, 6.5 percent higher than CAP, 1.4 percent higher than CBR, 5.5 percent higher than iDedup for the Fslhome dataset, and 3.5 percent higher than CAP, 1.1 percent higher than CBR, 0.7 percent higher than iDedup, 9.7 percent higher than HAR for the MAC incremental snapshots. In the meantime, FGDEFRAG rewrites 70, 25.6 and 25.2 percent less data than CAP, CBR and iDedup for the MAC full snapshots, 70.6, 16.4 and 67.3 percent less data than CAP, CBR and iDedup for the Fslhome dataset, and 65.7, 47.8, 87.7, 30 percent less data than CAP, CBR, HAR, iDedup for the MAC incremental snapshots. 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 (Section 2.2.1) and thus, it rewrites much few redundant chunks to disks. In addition, for the two baseline approaches based on non-defragmentation, Exact removes all of the redundant data while Sparse keeps some of the redundant data. When compared to FGDEFRAG, Sparse keeps as much as 13.8 percent of redundant data, causing the deduplication ratio to be lower by as much as 13.8 percent.

TABLE 2
Comparison Between FGDEFRAG and the Baseline Defragmentation Approaches (CAP, CBR, HAR and iDedup) and Baseline Non-Defragmentation Approach (Exact, Sparse) in Terms of the Deduplication Ratio and Amount of the Rewritten Data for All the Backup Versions of MAC Snapshots and Fslhome Dataset

	Deduplication Ratio (percentage)			Rewritten Data (GB)		
	MAC		fslhome	MAC		fslhome
	Full	Inc.		Full	Inc.	
FGDEFRAG	96.4	96	91.5	163	3.5	112
CAP	90	92.5	85	542	10.2	380.8
CBR	95.5	94.9	90.1	219	6.7	134
HAR	98	86.3	93	50	28.5	88.2
iDedup	95.5	95.3	86	218	5	343
Exact	99	97.2	93.5	0	0	0
Sparse	84.9	82.2	80.5	884	44	728

For iDedup, it gains very high deduplication ratios for both MAC full and MAC incremental snapshots. This is because in iDedup, it removes all of the duplicate sequences that is not less than 4 chunks, and thus most of the redundant data can be removed by iDedup since MAC snapshots have only a few duplicate sequences that are less than 4 chunks. But for Fslhome dataset, iDedup has a much lower deduplication ratio, which indicates that large amount of duplicate sequences whose length is less than 4 chunks exist in Fslhome dataset. In other words, iDedup may not suit for such kind of workload.

Moreover, FGDEFRAG underperforms HAR in deduplication ratios for the full snapshots of both datasets, even though it also reduces false positive fragmental data as described in Section 2.2.1. The main reason is that HAR has more 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. In the cases of incremental snapshots, HAR gets a much lower deduplication ratio. This is because in incremental backup streams, the backup data is much less than that in the full backup streams, and thus HAR identifies more containers as fragmental containers with a much higher false positives, and rewrites more fragmental chunks leading to a lower deduplication ratios than other defragmentation approaches.



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

Fig. 9. Comparison between FGDEFRAG and the baseline approaches in restore performance with the last 10 versions of the MAC full snapshots dataset. Speedup represents the restore performance normalized by that of the Exact+LRU approach.

4.3 Restore Performance

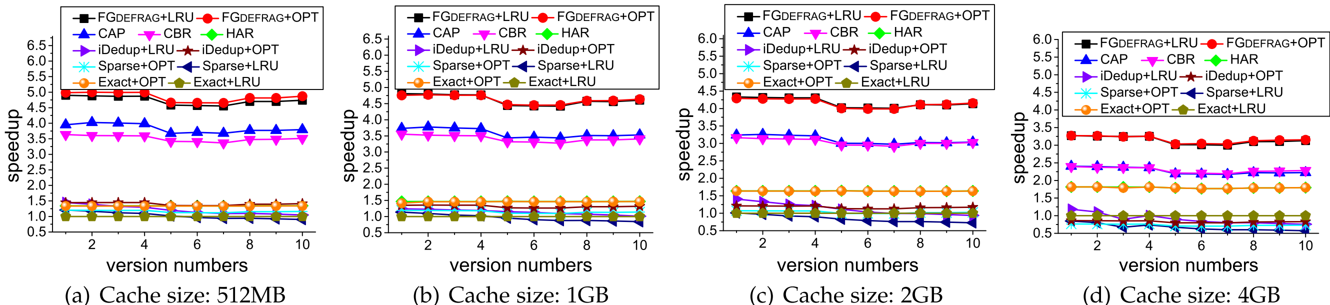
Restore performance is the most important performance metric for measuring the effectiveness of defragmentation approaches. In this section, we compare the restore performances between FGDEFRAG and the defragmentation baseline approaches (CAP, CBR, HAR and iDedup) and the non-defragmentation baseline approach (Exact and Sparse). For CAP, CBR and HAR, various cache algorithms are employed to improve the restore performance, For example, CAP uses Assembly Area [8]; HAR uses OPT [4]; and CBR uses LFK [7]. For the FGDEFRAG, iDedup and non-defragmentation baseline approaches, both LRU and OPT cache algorithms are implemented, i.e., FGDEFRAG+LRU/OPT, iDedup+LRU/OPT, Exact+LRU/OPT and Sparse+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. iDedup, on the other hand, evicts the reading duplicate sequences while Exact and Sparse (non-defragmentation baseline approaches) evicts the reading containers.

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. As a side note, while restoring an incremental backup version of each dataset, we generate the correct access order of the storage units by synthesizing the desired correct backup version of each file from the last full backup data, the current incremental backup data, and the incremental backups between the last full backup and the current incremental backup data.

4.3.1 Full Snapshots Backup

Figs. 9 and 10 compare the restore performance between FGDEFRAG and the baseline approaches to process MAC full snapshots and Fslhome datasets, respectively. As shown, FGDEFRAG outperforms all the baseline approaches consistently in all cases. For the MAC full snapshots dataset, FGDEFRAGE, on average, outperforms CAP, CBR, HAR and iDedup by 60, 20.2, 176, and 217 percent, respectively when the cache size is 512 MB; 63, 19.3, 116 and 238 percent, respectively when the cache size is 1 GB, and 62, 20, 23 and 236 percent, respectively when the cache size is 2 GB. For the Fslhome dataset, FGDEFRAG outperforms CAP, CBR, HAR and iDedup by 27, 38.2, 262 and 319 percent, respectively with a 512 MB cache; 30, 37.3, 217 and 306 percent with a 1 GB cache; 35, 38.3, 159 and 305 percent with a 2 GB cache; and 43, 39.2, 76, 329 percent with a 4 GB 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



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

Fig. 10. 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 Exact+LRU approach.

TABLE 3
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 Full Snapshots and Incremental Snapshots Datasets with a 1GB Cache

Approaches	# of disk seeks		reading data(GB)	
	Full	Inc.	Full	Inc.
iDedup+LRU	165864	254071	52.29	168.65
iDedup+OPT	161054	258050	51.08	165.68
Exact+LRU	72402	77681	303.68	325.82
Exact+OPT	55346	57307	232.24	240.36
Sparse+LRU	57370	81860	240.63	343.35
Sparse+OPT	46903	79883	193.33	335.06
HAR	29420	59023	123.4	247.56
CAP	22167	55343	92.98	232.13
CBR	16082	48090	67.13	225.76
FGDEFRAG+LRU	13228	32071	59.26	215.85
FGDEFRAG+OPT	12790	29668	57.43	209.73

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 3, leading to much higher restore performance.

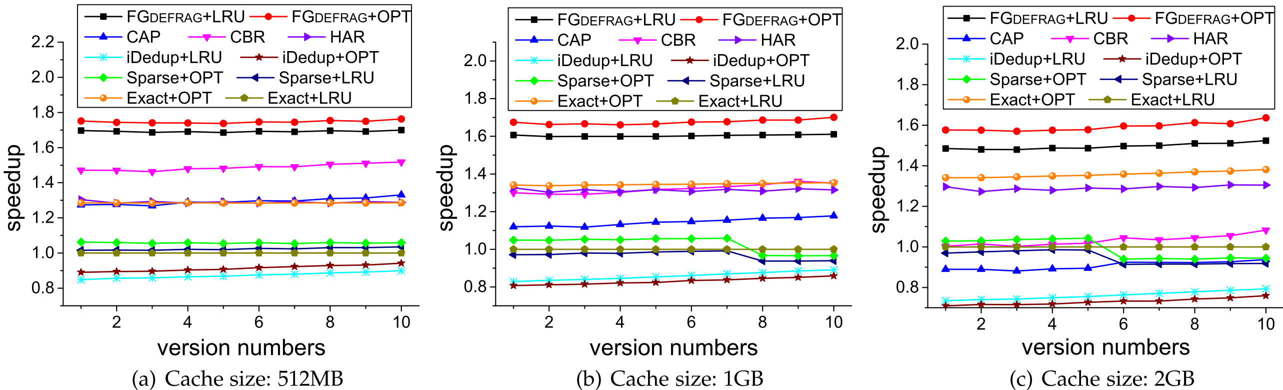
Since HAR misses identifying and rewriting local fragmental chunks for restore performance optimizations (as described in Section 4.2), its performance is only comparable to that of the baseline non-defragmentation approach Exact, which uses the OPT cache replacement algorithm when the cache size is either 512 MB, 1 GB, 2 GB or 4 *GB* when it is used to process the Fslhome dataset. For the MAC full snapshot dataset, it has the lowest restore performance among the three baseline defragmentation approaches when the cache size is 512 MB and 1 GB 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 2 GB 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 full snapshot and Fslhome datasets only require a disk space of 63 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.

In the case of iDedup, since there are many short duplicate sequences existing in the backup datasets, it needs large amounts of disk seeks to locate those short sequences. However, reading such short sequences each time cannot amortize the extra disk overhead; therefore, it has the lowest restore performance among all the defragmentation approaches. Table 3 shows that iDedup needs about ten times the amount of disk seeks of FGDEFRAG. Even though it can read all the referenced chunks in the duplicate sequence sequentially with a smallest amount of reading data, it still shows only one quarter to one third of the restore performance of FGDEFRAG. These results clearly indicate that iDedup is not appropriate for the data intensive backup systems that focus more on the throughput for massive data reads/writes rather than the data latency for small data reads/writes for primary storage workloads.

Moreover, all the defragmentation approaches designed for backup workloads, including FGDEFRAG, CAP, CBR, and HAR, outperform Sparse for all the cases. This is because Sparse does not locate the fragmented data and rewrite them to improve the restore performance. Its major goal is to accelerate the deduplication speed through keeping some redundant data not removed. Thus Sparse has a lower restore performance than all the defragmentation approaches designed to achieve restore performance improvements.

4.3.2 Incremental Snapshots Backup

Fig. 11 compares the restore performances of FGDEFRAG and those of the baseline approaches to process MAC incremental snapshots. As the same with the full snapshots, FGDEFRAG consistently outperforms all of the baseline approaches. On average, FGDEFRAGE outperforms CAP, CBR, HAR and



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

Fig. 11. Comparison between FGDEFRAG and the baseline approaches in restore performance with the last 10 versions of the MAC Incremental snapshots dataset. Speedup represents the restore performance normalized by that of the Exact+LRU approach.

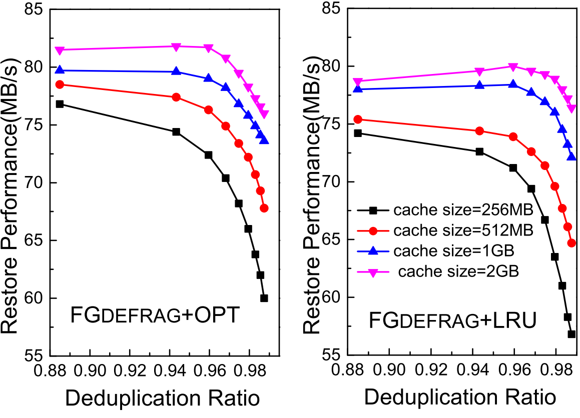


Fig. 12. Sensitivity of the restore performance and deduplication ratio to the value of the bandwidth threshold factor N. The restore performance is the average value of the last 20 versions of the MAC full snapshot dataset. The 9 data points on each curve represent the corresponding restore performance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

iDedup by 32, 14.1, 38, and 87 percent, respectively when the cache size is 512 MB; 45, 26.2, 30 and 101 percent, respectively when the cache size is 1 GB; and 75, 51.2, 25, and 118 percent respectively when the cache size is 2 GB. This performance gain not only comes from the accurate identification and rewriting of the fragmental chunks as that for full backups, but also benefits from the variable-sized data segments used for incremental backups to find the fragmental chunks.

However, different from the full snapshots, all of the defragmentation solutions does not show as much significant improvement as for the full snapshots on the restore performances over the non-defragmentation baseline approaches. The main reason is that restoring incremental snapshots requires reading the latest full backup and all the subsequent incremental backups to generate the desired correct backup version. Thus, the chunks required for each restore are distributed across many backups, and so too the fragmental chunks. Independently and solely identifying the fragmental chunks in a single incremental backup stream cannot fully identify the fragmental chunks to improve the restore performance. As our future work, we hope to propose a more flexible and dynamic defragmentation solution to improve the restore performance for incremental backups.

As a side note, compared to other defragmentation approaches, FGDEFRAG has added an extra group table to store the address information of each group. Our experimental results show that this group table is only 447 KB for the MAC full snapshot, 327 KB for the MAC Incremental snapshot, and 1.27 MB for the Fslhome dataset, which is negligibly small compared to the total size of 63, 45, and 191 GB unique chunks stored on disks for the MAC full snapshot, incremental snapshot, and Fslhome dataset respectively, and the restore cache whose capacity ranges from 256 MB to 4 GB.

4.4 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. 12 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 full snapshot dataset.

As seen from the results, the deduplication ratio increases with N, especially from 2 to 4 when it increases from 88 to 94 percent. On the other hand, the restore performance decreases significantly as N increases. For example, under the LRU algorithm with a 256 MB cache, the restore performance is reduced by about 30 percent 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 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.

5 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 identification and effective exploitation of chunk locality enable it to improve both the restore performance and deduplication ratio. Our experimental results show that FGDEFRAG outperforms four state-of-the-art defragmentation schemes, CAP, CBR, HAR and iDedup in restore performance by 27 to 75 percent, 14 to 51 percent, 23 to 262 percent, and 87 to 329 percent, respectively. It also reduces the amount of rewritten data by 25 to 87 percent. In terms of deduplication ratios for full snapshots, FGDEFRAG also outperforms CAP and CBR but slightly underperforms HAR, because HAR identifies the fragmental chunks globally but at the expense of missed identification of some local fragmental chunks; and therefore, HAR rewrites fewer redundant chunks, leading to slightly higher deduplication ratio. However, FGDEFRAG does not show as much significant performance improvement when used to process incremental backups as compared to that of full backups. We plan to focus on addressing this special fragmentation problem for incremental backup workloads in our future work.

Acknowledgments

A preliminary version of this work was presented at the 33rd International Conference on Massive Storage Systems and Technology (MSST 2017) and we have made substantial changes in this manuscript. This work is supported in part by National Natural Science Foundation of China (NSFC) under Grant No.61402061, Chongqing Basic and Frontier Research

Project of China under Grant No.cstc2016jcyjA0274, US National Science Foundation under Grant No.CCF-1704504 and 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," HP Laboratories, Palo Alto, CA, USA, Tech. Rep. HPL-2009-10R2, 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 *Proc. 7th Conf. File Storage Technol.*, Feb. 2009, pp. 197–210.

[3] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, J. Liu, W. Xia, F. Huang, and Q. Liu, "Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 3, pp. 855–868, Mar. 2016.

[4] 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 Conf. USENIX Annu. Tech. Conf.*, Jun. 2014, pp. 181–192.

[5] 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 *Proc. 8th ACM Int. Syst. Storage Conf.*, Feb. 2015, Art. no. 17.

[6] M. Kaczmarczyk, M. Barczynski, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in *Proc. 5th Annu. Int. Syst. Storage Conf.*, Jun. 2012, Art. no. 15.

[7] M. Kaczmarczyk and C. Dubnicki, "Reducing fragmentation impact with forward knowledge in backup systems with deduplication," in *Proc. 8th ACM Int. Syst. Storage Conf.*, Jun. 2015, Art. no. 17.

[8] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proc. 11th USENIX Conf. File Storage Technol.*, Feb. 2013, pp. 183–198.

[9] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Campbell, "Sparse indexing: Large scale, inline deduplication using sampling and locality," in *Proc. 7th Conf. File Storage Technol.*, Feb. 2009, pp. 111–123.

[10] Y. Nam, G. ParkG. Lu, W. Xiao, and D. H. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Sep. 2011, pp. 581–586.

[11] Y. J. Nam, D. Park, and D. H. Du, "Assuring demanded read performance of data deduplication storage with backup datasets," in *Proc. IEEE 20th Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, Aug. 2012, pp. 201–208.

[12] NIST, "Secure hash standard," NIST, Gaithersburg, MD, USA, Tech. Rep. 180-1, May 1993.

[13] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in *Proc. Conf. File Storage Technol.*, Jan. 2002, pp. 1–13.

[14] M. O. Rabin, "Fingerprinting by random polynomials," Center Res. Comput. Technol., Harvard Univ., Cambridge, MA, USA, Tech. Rep. TR-15-81, 1981.

[15] P. Shilane, R. Chitloor, and U. K. Jonnala, "99 deduplication problems," in *Proc. 8th USENIX Conf. Hot Topics Storage File Syst.*, Jun. 2016, pp. 86–90.

[16] K. Srinivasan, T. Bisson, G. Goodson, and Y. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proc. 10th USENIX Conf. File Storage Technol.*, Feb. 2012, pp. 24–24.

[17] Traces and Snapshots Public Archive. (2014). [Online]. Available: http://tracer.filesystems.org/

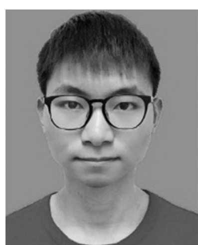
[18] J. Wu, Y. Hua, P. Zuo, and Y. Sun, "A cost-efficient rewriting scheme to improve restore performance in deduplication systems," in *Proc. Int. Conf. Massive Storage Syst. Technol.*, May 2017, pp. 1–12.

[19] 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 *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, Jun. 2012, pp. 285–298.

[20] L. L. You, K. T. Pollack, and D. D. E. Long, "Deep store: An archival storage system architecture," in *Proc. Int. Conf. Data Eng.*, Apr. 2005, pp. 804–815.

[21] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proc. 6th USENIX Conf. File Storage Technol.*, Feb. 2008, Art. no. 18.

Yujuan Tan received the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012. She now works as an associate professor with the College of Computer Science, Chongqing University, China. Her main research interests focus on massive storage system, data backup and recovery, data deduplication, transactional memory. She has published several papers in international conferences and journals, including FAST, MSST, IPDPS, ICPP, HotStorage, ISPA, HiPEAC, the *ACM Transactions on Architecture and Code Optimization*, the *Cluster Computing* and the *IEEE Access*. She is a member of the IEEE.



Baiping Wang is currently working toward the master's degree majoring in computer architecture at Chongqing University, Chongqing, China. His current research interests include data deduplication, non-volatile main memory storage systems, hybrid memory systems.

Jian Wen received the MS degree from Chongqing Univeristy, Chongqing, China, in 2017. His main research interests include data deduplication and SMR storage systems. He has several papers in major journals and conferences including the *IEEE Transactions on Parallel and Distributed Systems*, MSST, etc.

Zhichao Yan received the PhD degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2012. He is working toward the PhD degree major in computer science with the University of Texas at Arlington. He then worked as a postdoc scholar with the Department of Computer Science and Engineering, University of Nebraska-Lincoln. Now he is an assistant researcher with the University of Texas at Arlington. His research interests focus on cloud storage, data deduplication, data compression, transactional memory, reliability-aware computing. He has published several papers in international conferences and journals, including HotStorage, IPDPS, ICPP, MSST, HiPEAC, and the *ACM Transactions on Architecture and Code Optimization*.

Hong Jiang received the BSc degree in computer engineering from the Huazhong University of Science and Technology, Wuhan, China, in 1982, the MASc degree in computer engineering from the University of Toronto, Toronto, Canada, in 1987, and the PhD degree in computer science from the Texas A&M University, College Station, Texas, in 1991. He is currently chair and Wendell H. Nedderman endowed professor of the Computer Science and Engineering Department, University of Texas at Arlington. Prior to joining UTA, he served as a program director at National Science Foundation (2013.1-2015.8) and he was with the University of Nebraska-Lincoln since 1991, where he was Willa Cather professor of computer science and engineering. He has graduated 16 PhD students who upon their graduations either landed academic tenure-track positions in PhD-granting US institutions or were employed by major US IT corporations. He has also supervised 15 post-doctoral fellows and visiting scholars. His present research interests include computer architecture, computer storage systems and parallel I/O, high-performance computing, big data computing, cloud computing, performance evaluation. He recently served as an associate editor of the *IEEE Transactions on Parallel and Distributed Systems*. He has more than 250 publications in major journals and international Conferences in these areas, including the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *Proceedings of IEEE*, the *ACM Transactions on Architecture and Code Optimization*, the *ACM Transactions on Storage*, the *Journal of Parallel and Distributed Computing*, ISCA, MICRO, USENIX ATC, FAST, EUROSYS, LISA, SIGMETRICS, ICDCS, IPDPS, MIDDLEWARE, OOPLAS, ECOOP, SC, ICS, HPDC, INFOCOM, ICPP, etc., and his research has been supported by NSF, DOD, and industry. He is a fellow of the IEEE, and member of the ACM.

Witawas Srisa-an received the PhD degree in computer science from the Illinois Institute of Technology, in 2002. He is currently an associate professor of computer science and engineering with the University of Nebraska-Lincoln. His research interest spans the area of programming languages, software engineering, operating systems, runtime systems, and security. He is a member of the IEEE.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.