Received: 4 March 2021 | Revised: 18 August 2021 | Accepted: 10 October 2021

DOI: 10.1002/cpe.6706

RESEARCH ARTICLE

WILEY

Ensuring high reliability and performance with low space overhead for deduplicated and delta-compressed storage systems

Chunxue Zuo[1] | Fang Wang[1] | Mai Zheng[2] | Yuchong Hu[1] | Dan Feng[1]

[1]Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, School of Computer Science and Technology, Huazhong University of Science and Technology, Ministry of Education of China, Wuhan, China

[2]Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA

Correspondence

Fang Wang, Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System, School of Computer Science and Technology, Huazhong University of Science and Technology, Ministry of Education of China, Wuhan 430074, China.
Email: wangfang@hust.edu.cn

Funding information

Fundamental Research Funds for the Central Universities; National Defense Preliminary Research Project, Grant/Award Number: 31511010202; National Key R&D Program of China, Grant/Award Number: 2018YFB10033005; National Natural Science Foundation of China, Grant/Award Numbers: 61832020, 61772216; National Science and Technology Major Project, Grant/Award Number: 2017ZX01032-101

Abstract

Data deduplication is a widely used technique to remove duplicate data to reduce the storage overhead. However, deduplication typically cannot eliminate the redundancy among nonidentical but similar data chunks. To reduce the storage overhead further, delta compression is often applied to compress the post-deduplication data. While the two techniques are effective in saving storage space, they introduce complex references among data chunks, which inevitably undermines the system reliability and introduces fragmentation that may degrade the restore performance. In this paper, we observe that the delta compressed chunks (DCCs) are much smaller than regular chunks (non-DCCs). Also, most fragmentation caused by the base chunk of DCCs remain fragmented in consecutive backups. Based on these observations, we introduce a framework called RepEC$^+$, which combines replication and erasure coding and uses History-aware Delta Selection to ensure high reliability and restore performance. Specifically, RepEC$^+$ uses a delta-utilization-aware filter and a cooperative cache scheme (CCS) to maintain cache locality and avoid unnecessary container reads, respectively. Moreover, the system selectively performs delta compression by historical information to avoid cyclic fragmentation in consecutive backups. Experimental results based on four real-world datasets demonstrate that RepEC$^+$ significantly improves the restore performance by 58.3%–76.7% with a low storage overhead.

KEYWORDS

data deduplication, delta compression, fragmentation, reliability, restore performance, storage systems

1 | INTRODUCTION

With the explosive growth of digital data in recent years, the demand for storage space has been ever-increasing. For instance, International Data Corporation (IDC) reports that the amount of digital data distributed in the whole world has increased rapidly, and will reach 175ZB by 2025.[1] Consequently, how to store the huge volume of data efficiently becomes an increasingly challenging problem, especially for distributed storage systems that need to manage large-scale datasets.

To address the challenge, data deduplication has been proposed to eliminate redundant data in distributed storage systems.[2,3] A typical chunk-level deduplication process divides an incoming data stream into fixed-size or variable-length chunks,[4,5] and identifies unique chunks by

 © 2021 John Wiley & Sons, Ltd.

TABLE 1 Usage of deduplication and delta compression in five representative storage systems. "Yes" means the technique is used in the corresponding system

Systems	Deduplication	Delta compression
LBFS[10]	Yes	—
SIDC11, [11]	Yes	Yes
Neptune[12]	Yes	Yes
QuickSync[13]	Yes	Yes
Dropbox[14]	Yes	Yes

examining chunk fingerprints. A chunk fingerprint is a fixed-length message digest of the chunk content, which is calculated using cryptographic functions (e.g., SHA1 or MD5). All fingerprints are stored in a fingerprint index store.[6] During the backup procedure, the fingerprint of every incoming chunk is compared with the fingerprints stored in the fingerprint index previously. If a match is found, the incoming chunk is considered to be duplicate and thus will not be physically stored. Otherwise, the chunk is unique and will be written to a currently open storage unit called "container." During the restore procedure, containers are read from storage nodes to a restore cache to provide the required chunks of a file.[7] Such deduplication technique can eliminate duplicate chunks effectively. However, it cannot handle the redundancy among nonduplicate but similar chunks.[8,9]

To address the limitation, practical backup storage systems often integrate delta compression with deduplication to further reduce the storage overhead.[11-15] Specifically, after the chunk-level deduplication, delta compression detects whether there is any matched string between the incoming nonduplicate chunk and the chunks that are already stored in the system. If so, the incoming nonduplicate chunk has a similar chunk in the system. In this case, the nonduplicate chunk and its corresponding similar chunk are encoded to generate a difference called *delta compressed chunk* (DCC). The corresponding similar chunk is called the *base chunk* of the nonduplicate chunk. By definition, both the base chunk and the DCC are required to restore the nonduplicate chunk. Such a combination of deduplication and delta compression has been widely deployed in state-of-the-art systems. As shown in Table 1, four out of the five representative storage systems use both deduplication and delta compression together to minimize the storage footprint.[11-14]

However, applying delta compression to a deduplicated storage system introduces additional complexity that may affect the system reliability negatively.[16] For example, due to the DCCs created by delta compression, the dependencies between chunks and files include not only "*share*" (i.e., multiple files sharing the same chunk) but also "*reference*" (i.e., a chunk being referenced by another chunk). When a chunk of a file is lost, all other files that share or refer to the lost chunk are in danger due to the complicated "share" and "reference" relationship.

To ensure the reliability, existing work use either replication[17] or erasure coding[18-20] to redundantly distribute data across multiple nodes, trading off storage space for reliability. Different from existing approaches which are agnostic to deduplication and delta compression, we observe that DCCs are much smaller than non-DCCs in typical deduplicated and delta compressed systems. Based on such observation, we propose a hybrid reliability scheme (HRS) that stores DCCs with replication for high performance and stores non-DCCs with erasure codes for less storage overhead.

While the hybrid scheme makes sense intuitively, it results in an inefficient cache problem that consists of two subproblems, namely *cache locality problem* and *cache miss read*. These need to be addressed properly in order to make the hybrid scheme practical:

First, the random placement of DCC replicas in containers may hurt the cache locality during the failure recovery. To restore a lost DCC, it is necessary to read the container that hosts the DCCs replica. Because DCCs and non-DCCs are mixed in the container, not all chunks loaded to the restore cache are useful for recovering the lost DCC, which leads to poor cache usage and thus affects the recovery performance.

Second, the separation of the decoding cache and the restore cache may lead to unnecessary read operations. In a typical deduplicated and delta compressed storage system, the decoding recovery and the file restore are two separate procedures supported by the decoding cache and the restore cache, respectively. Correspondingly, the same container may need to be read to the two caches repeatedly during the recovery, which hurts the performance.

To address challenges, we propose RepEC$^+$, an efficient hybrid reliability mechanism to provide efficient storage and high restore performance for deduplicated and delta compressed storage systems. Specifically, to address the cache locality problem, we utilize a delta-utilization-aware filter to select and store containers with replication based on the DCCs percentage of the containers, which enables maintaining the cache locality. To address the cache miss read, we use a cooperative cache which integrates the information of a decoding cache into a restore cache during the recovery. In doing so, the cache locality is improved further, and the number of container reads is reduced significantly.

In addition, we observe that both deduplication and delta compression may incur fragmentation problems, which may degrade the restore performance seriously. While existing rewriting algorithms (e.g., CBR,[21] CAP,[22] History-Aware Rewriting Algorithm (HAR)[23]) may alleviate the fragmentation caused by deduplication, they cannot handle the fragmentation stemming from the base chunks of DCCs. To address this problem, we

design a technique called History-aware Delta Selection (RepEC-HDS), which exploits historical information to selectively perform delta compression between two similar chunks. Experimental results show that RepEC-HDS improves the restore performance further with negligible impact on redundancy elimination.

In summary, the main contributions of this paper are as follows:

(1) We observe that DCCs are much smaller than non-DCCs in typical deduplicated and delta compressed storage systems. We also reveal that most fragmentation caused by the base chunk of DCCs remain fragmented in consecutive backups(detailed in Section 2).
(2) We propose an efficient hybrid reliability mechanism called RepEC$^+$, which introduces a delta-utilization-aware filter to select and replicate proper containers based on the percentage of DCCs in the containers. Moreover, RepEC$^+$ employs a cooperative cache to enable a restore cache to share the containers from a decoding cache.
(3) We develop a RepEC-HDS scheme which further improves the restore performance.
(4) Our experimental results show that, compared with the existing reliability schemes, RepEC$^+$ significantly improves the restore performance by 58.3%–76.7% with a low storage overhead.

The rest of this paper is organized as follows. Section 2 presents the background, motivation and and related work of our research. Section 3 elaborates on the architectural design and implementation of our RepEC$^+$ scheme. Section 4 presents experimental evaluations with four datasets. Section 5 concludes the paper.

2 | BACKGROUND, MOTIVATION, AND RELATED WORK

In this section, we discuss the background of deduplication and delta compression, reliability, fragmentation and give motivation to present the problem of our research. We also discuss related work in the corresponding subsections to further motivate our work.

2.1 | Deduplication and delta compression

Data deduplication is an important compression technology to save storage space by removing duplicate chunks in modern backup systems.[24,25] A typical deduplication process begins with dividing an inputting data stream to relatively small fixed-size or variable size data chunks,[5] and then computes each chunk's fingerprint using cryptographic hash algorithms (e.g., SHA-1, SHA-256, MD5)[7] to distinguish the uniqueness of chunks. If any of fingerprints is matched with previously stored fingerprints in the storage systems, the chunk is deemed to be duplicate. Otherwise, it is a unique chunk and needs to be stored in the systems. Meister et al.[26] make a detailed study on four HPC centers and adopt deduplication to improve the storage utilization. Wallace et al.[27] achieve high deduplication efficiency by analyzing statistics and content metadata collected from a large set of EMC Data Domain backup systems. Sun et al.[28] analyze several snapshots of each individual user, and find that we can group users together to maximize deduplication.

Delta compression is another compression technology to remove non-redundant but similar data to further reduce the storage space.[29] Some systems, such as Neptune, SIDC and Quicksync, adopt deduplication and delta compression to improve the storage utilization. Neptune[12] proposes a remote backup framework that uses local deduplication and approximate delta compression to implement the data reduction, thus reducing the remote communication overheads. SIDC[11] deploys stream-informed delta compression based on the deduplication. Quicksync[13] adaptively selects the deduplication strategy according to network-aware chunker, implements delta encoding and designs a batched syncer to improve the sync efficiency for the mobile cloud. In addition to the above systems, Dropbox[14] as a commercial cloud service has the capability of deduplication and delta compression. In this work, we use the *Xdelta* compression algorithm. Assume that chunk A is similar to chunk B, Xdelta[30] uses the COPY/INSERT instructions to generate the difference which is called the "delta" between chunk A and chunk B. Chunk B is referred to as the *base chunk* of Chunk A. Even though the storage system does not store the content of Chunk A, Chunk A can be easily restored by decoding the delta and its base Chunk B.

2.2 | Reliability scheme

The deduplication-based storage system reduces storage space by storing one copy of redundant data at a potential risk of reducing reliability.[18] To improve reliability, replication-based scheme and erasure codes have been proposed. Replication-based scheme aims to retain several copies of each data chunk in the deduplication-based storage systems.[17] This method provides high reliability and good restore performance in the events of failures, but it introduces significant storage overheads, for example, in a three-way replication system, it consumes three times storage space. Hence, replication-based scheme is not space-efficient for the large-scale storage systems. Compared with replication, erasure code is more space-efficient and is commonly used to ensure data reliability.[18-20] R-ADMAD[18] exploits ECC codes to encode the fix-sized objects and distributes them across

the storage nodes. DAC[31] exploits the data reference characteristic to combine replication and erasure coding schemes in the cloud storage systems. Xiao et al.[32] propose CodePlugin to research how to efficiently integrate the inline deduplication technology into the erasure coding in the cloud environment. Wu et al.[33] propose a PFP scheme to compute the parity for a group of chunks or a file to provide redundancy protection for all files in order to improve the reliability of deduplication-based storage systems. Rozier et al.[34] put forward a framework to analyze the reliability of deduplication-based storage systems. Based on the observation of redundancy characteristics, Fu et al.[35] propose a simulation framework and reliability metrics to analyze storage system reliability with and without deduplication. Li et al.[20] present and analyze the combination of deduplication and erasure coding to evaluate the key-value store system reliability. We take an (k,m) erasure code for example. First, we divide data into k data chunks and then encode them to generate $m$parity chunks ((k+m) = n total chunks). The set of these n chunks stored in n storage nodes is called a *stripe*. If any of the n nodes fail in the deduplication-based storage systems, the containers belonging to the failed node become unavailable. To recover a container on a failed node requires to read a total of k data and parity containers from the same stripe, degrading restore performance.

The augmentation of delta compression to deduplicated storage systems increases the probability of data failure, as the loss of a chunk not only affects files sharing the lost chunk, but also damages the chunks whose base chunk is the lost chunk. This is because the relationship between chunks or files contains not only "share" but also "reference" in the deduplicated and delta compressed storage system. Figure 1 gives an illustrative example to show the relationship between multiple files. In the figure, both files 1 and 2 share chunk B. In the deduplication-based systems, if chunk B is lost, we cannot restore files 1 and 2 because they share the same chunk B. However, in the deduplicated and delta compressed systems, if the chunk B is lost, all three files cannot be restored. It is mainly because chunk B* in file 3 is similar to chunk B in file 2. Thus, the relationship between chunk B and B* is referred to as "reference." Based on the above analysis, we can see that ensuring data reliability is critical for deduplicated and delta-compressed storage systems. Thus, we need a carefully designed mechanism to ensure data reliability in deduplicated and delta-compressed storage systems.

Different from the storage objects of deduplication-based storage systems, the deduplicated and delta compressed storage system has two types of chunks. One is DCC, and the other is non-DCC (we call regular chunk in this paper). As delta compression eliminates redundancy at finer granularities than data deduplication (i.e., byte level vs. chunk level), we observe that delta compressed chunks are far smaller than regular chunks. To illustrate the characteristic of two types of chunks, we use three datasets (detailed in Section 4.1) to evaluate the average DCC size and regular chunk size. Figure 2 shows the result. In Linux, GCC, and RDB datasets, the DCCs are smaller than regular chunks by 18×, 9×, and 5× on average, respectively. Intuitively, to ensure data reliability, the replication of DCCs can provide higher restore performance at the cost of minimal storage overhead than that of regular chunks.

To further investigate the characteristics of data chunks, we calculate the chunk count and the storage space occupied by each backup in Figure 3. For the Linux, we observe that DCCs and regular chunks, respectively, account for 93% and 7% of the total chunk count in the deduplicated and delta compressed storage systems. However, in Figure 3A, we observe that only a small number of regular chunks almost account for the same storage space as the DCCs. For the GCC, 25% regular chunks occupy about 75% of total storage space. These results suggest that a very small

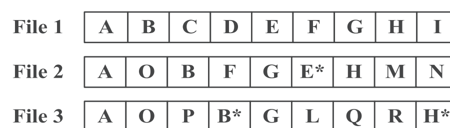

FIGURE 1 The relationship between multiple files

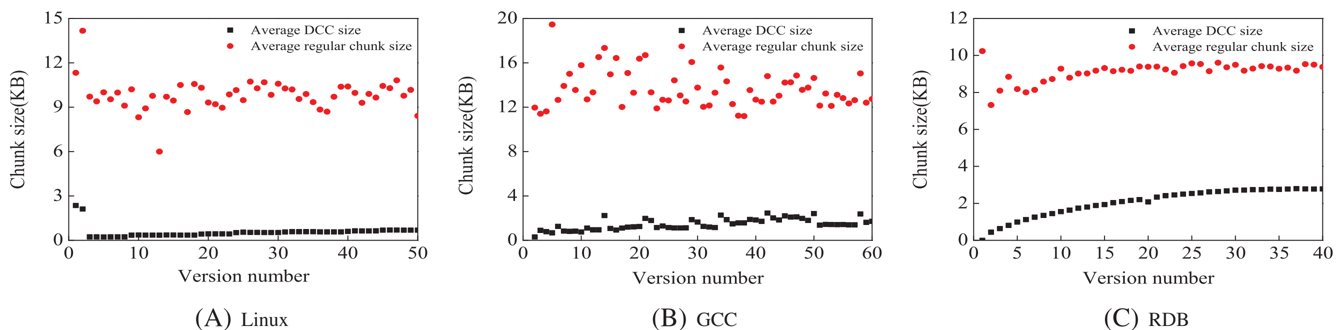

FIGURE 2 The comparison between average delta compressed chunks size and average regular chunk size of three datasets. (A) Linux; (B) GCC; (C) RDB

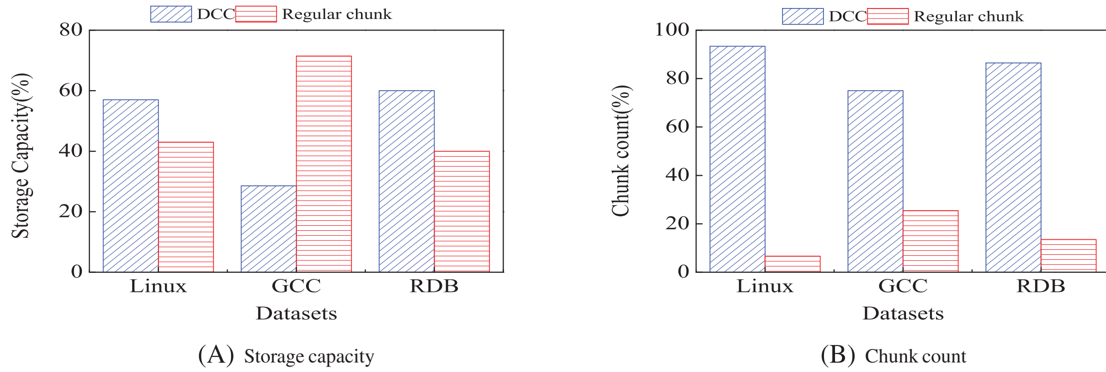

FIGURE 3 The percentage of delta compressed chunks and regular chunks in terms of storage capacity and chunk count for the three datasets. (A) Storage capacity; (B) Chunk count

number of large chunks (e.g., regular chunks) occupy large percentage of storage space, while the large number of small DCC chunks only takes small amount of storage space. Motivated by above observations, one naive solution is to design a hybrid reliability mechanism that uses replication method to replicate DCCs and erasure code to encode regular chunks to guarantee system reliability. However, such reliability solution results in the restore performance degradation, which is caused by inefficient cache and fragmentation problems. We will discuss these two problems in the next section.

2.3 | Inefficient cache problem

Inefficient cache problem has two sources. One is cache locality problem, the other is cache miss read. Next, we will conduct an in-depth discussion for this problem.

(1) *Cache locality problem*: The random placement of DCC replicas in containers may harm the restore cache locality during the failure recovery. Figure 4 gives an illustrative example of a deduplicated and delta compressed storage system using (3,1) erasure code. As shown in the figure, both Files 1 and 2 contain nine chunks. After duplicating File 1, its nine unique chunks are stored in containers C0, C1, and C2 on node0, node1, and node2, respectively. In an erasure coding scheme, we encode three containers to form 1 parity chunk P0, and then distribute the three data containers and the resultant parity container to four nodes in a round-robin way. As illustrated in Figure 4, File 2 has four unique chunks (i.e., J, K, M, N) and two similar blocks (i.e., A*, D*) to File 1, which are stored in containers C3 and C4 in the next stripe. In a HRS, since chunk A_d and D_d are small delta encoded chunks, we make replicas of them in container C4 as shown in Figure 4B. When node1 fails, we can read the container C4 to directly obtain A_d to restore the chunk A* of file 2. However, we find that the read of C4 is not cost-efficient. This is because chunks M and N will be not used recently except for obtaining a copy of A_d in container C4, which destroys the cache locality. (2) *Cache miss read*: The separation of the decoding cache and

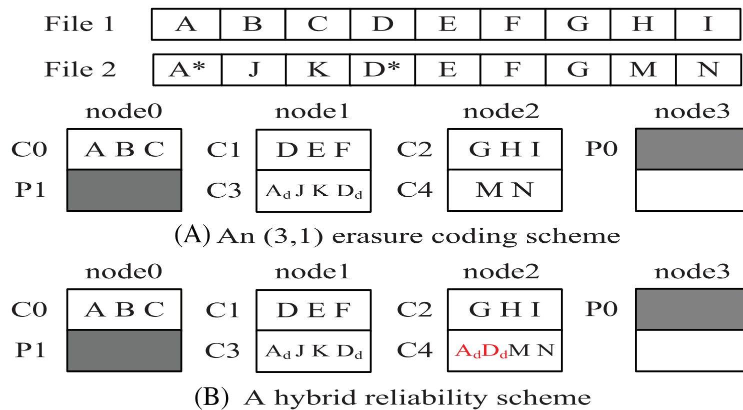

FIGURE 4 The placement of delta compressed chunks' copies impairs cache locality. Cx denotes a container and a character denotes a Chunk. A character with a subscripted "*d*" refers to a delta encoded chunk. (A) An (3.1) erasure coding scheme; (B) A hybrid reliability scheme

the restore cache may lead to unnecessary read operations. In a typical deduplicated and delta compressed storage system, the decoding recovery and the file restore are two separate procedures supported by the decoding cache and the restore cache, respectively. This is because multiple containers from the same stripe are read to the decoding cache during the decoding recovery, while multiple containers from different stripes are read to the restore cache during the file recovery. As a result, the container reads for the two processes might be overlapped, which hurts the restore performance.

2.4 Fragmentation problem

Data reduction technologies, such as data deduplication and delta compression, cause fragmentation problem.[9,23] Figure 5 shows how the fragmentation arises in deduplicated and delta-compressed storage systems. File 1 has no duplicate chunks and thus 10 unique chunks are stored in the containers C1 and C2. File 2 contains four duplicate chunks and two similar chunks. After backing up File 2 is finished, only four unique chunks and two DCCs are stored in the container C3 because duplicate chunks has been stored in the first 2 containers and will not be stored. Likewise, after backing up File 3, unique chunks O, P, R, S and DCCs B_d, G_d, K_d are stored in the container C4. We note that all chunks of File 1 are aggregated in the previous containers C1 and C2, but the chunks of File 3 are scattered across four containers. As the number of backups increases, the chunks of later files are scattered across more and more containers. This phenomenon is known as fragmentation problem. As shown in Figure 5, the recovery of File 3 needs to read four containers, one more than that of File 2. This is because the read of container C2 is not efficient for File 3, only two chunks F and G in the container C2 are used. Such container is called a fragmented container. We define a *container's utilization* for a file as the fraction of its chunks referenced by the file. If a container's utilization is smaller than 50%, the container is regarded as a fragmented container for the file.[23] In the fragmented container C2, chunks F and G referenced by File 3 are regarded as fragmentation. We call chunk F that stems from a duplicate chunk a *normal fragmentation* and chunk G that stems from a base chunk of a DCC a *base fragmentation*.

Previous studies such as CBR,[21] CAP,[22] HAR,[23] mainly paid attention to rewriting fragmented chunks to reduce the fragmentation caused by deduplication. Except for above algorithms, some methods, such as iDeDup,[36] RevDedup[37] focus on eliminating fragmentation caused by chunk-level deduplication in different systems, including primary storage systems, cloud storage systems, etc. SDC[9] alleviates the base fragmentation by simulating a restore cache to identify and selectively perform delta compression among similar chunks. While existing rewriting algorithms may alleviate the normal fragmentation, such as chunk F, they cannot handle the base fragmentation. For base fragmentation, we have observed that they remain fragmented in consecutive backups. We call this phenomenon *cyclic fragmentation*.

Figure 6 gives an example to explain the reason for this phenomenon in incremental backups. After backing up File 1 and File 2, the chunks of File 1 and File 2 are stored in the containers C1 and C2 as described above. Since chunks B*, G*, K*, and L* of File 3 are similar to chunks B, G of File 1 and chunks K, L of File 2, chunks K, L and G become base fragmentation after backing up File 3. Likewise, chunks B**, G**, K**, and L** of File 4 are similar to chunks B, G in File 1 and chunks K, L in File 2, chunks K, L, and G also become base fragmentation after backing up File 4. Since both File 3 and File 4 share three similar chunks, they also share similar base fragmentation. It can thus be seen that base-fragmented chunks remain fragmented in consecutive files. This is because consecutive backups are very similar in incremental backup.[4] Therefore, they share similar chunks, including their base fragmentation.

To clearly illustrate the same degree of base fragmentation among consecutive files, we first define *identically base-fragmented percentage*. Identically base-fragmented percentage is defined as the ratio of the number of identically base fragmentation between each file and previously stored files to the number of all the base fragmentation in previously stored files. Thus, a larger percentage means more identically base fragmentation among consecutive files. We use three real-world datasets, Linux, GCC, and RDB (detailed in Section 4) to conduct experiments. After backing up each file, we first record the fingerprints of the base-fragmented chunks for the file. Through matching the fingerprints of base fragmentation of the file with these recorded fingerprints, the identically base fragmentation will be identified during each backup. We then calculate the identically

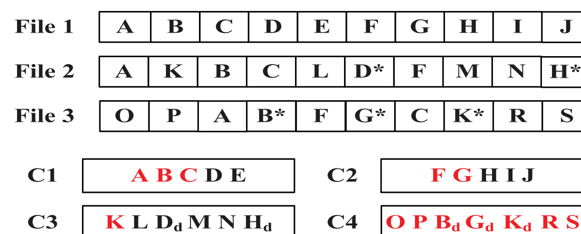

FIGURE 5 An example of two types of fragmentation arising among multiple files. The red chunk represents the chunks referenced by File 3 in each container

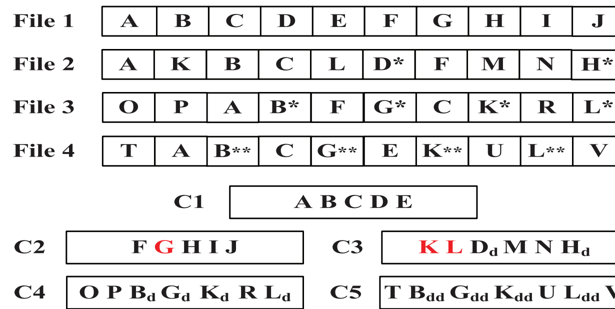

FIGURE 6 Base fragmentation G, K, L remain fragmented in continuously similar files (File 3 and File 4)

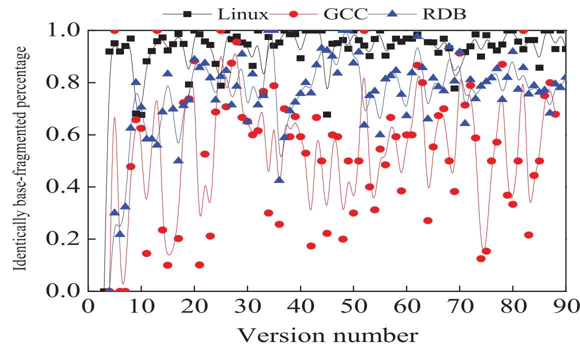

FIGURE 7 The percentage of base chunks for the three datasets

base-fragmented percentage of each file for three datasets and obtain the result as shown in Figure 7. For Linux, GCC, and RDB datasets, identically base-fragmented percentage is 87%, 56%, and 45% on average, respectively. It suggests that most of base fragmentation in the current file are nearly same as that in previous files for three datasets. Therefore, base fragmentation remain fragmented in consecutive files, which degrades restore performance.

The above analysis and observation motivate us to propose RepEC$^+$, a framework with high reliability and restore performance by optimizing cache and reducing cyclic fragmentation. During backups, RepEC$^+$ selectively replicate containers based on the delta utilization (denoted as DU) of each container to improve cache locality. During restores, RepEC$^+$ merges the information of a decoding cache and a restore cache to avoid repeatedly accessed containers. Due to our observation that base fragmentation remain fragmented in consecutive files, RepEC$^+$ identifies cyclic fragmentation through historical information and selectively performs delta compression for them. The design and implementation of RepEC$^+$ will be detailed in the next section.

3 | DESIGN AND IMPLEMENTATION

In this section, we first present the architecture of RepEC$^+$ and then give an description of its main modules.

3.1 | Architecture overview

RepEC$^+$ includes three key modules: redundancy detection, RepEC-HDS, RepEC-Core. Figure 8 illustrates the architecture overview of RepEC$^+$. Redundancy detection includes *duplicate identification* and *resemblance detection*. Duplicate identification first identifies an incoming chunk whether duplicate through inquiring the *index pool*, The index pool records all previously stored fingerprints of chunks. The chunk which has a same fingerprint in the index pool is duplicate and is marked as *duplicate*. On the contrary, the chunk is unique and will be represented by *unique*. After duplicate identification, these unique chunks need to be detected redundancy further and thus will be sent to the next structure *resemblance detection*. The process of resemblance detection is the same as that of traditional super-feature methods. Resemblance detection first computes chunks' features like "fingerprint" by Rabin fingerprints and then uses these features to generate *super-features*. These super-features

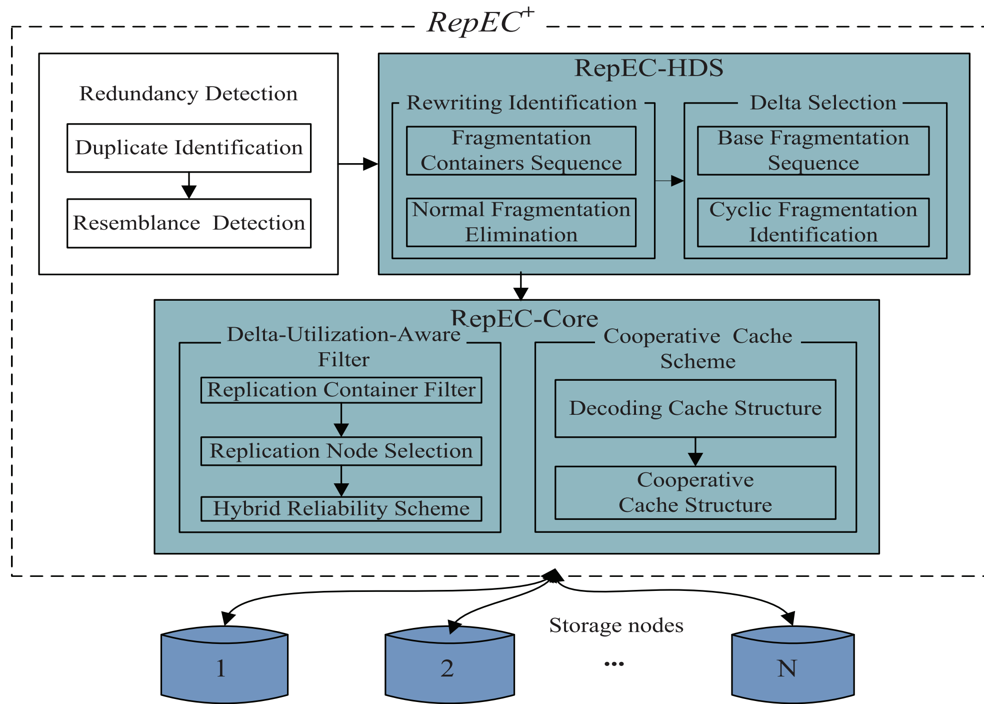

FIGURE 8 Architecture overview of RepEC+

comprises a *sketch*. A chunk with a matched super-feature in the sketch is regarded as a similar chunk and the correspondingly matched chunk is its base chunk.

RepEC-HDS includes *rewriting identification* and *delta selection*. Rewriting identification aims to generate fragmentation information of each backup and eliminate normal fragmentation caused by data deduplication. Delta selection uses the fragmentation information collected by rewriting identification to identify base-fragmented chunks and selectively carry out delta compression. Therefore, delta selection and rewriting identification share the fragmentation information. Since HAR[23] is able to generate the historical information, we implement rewriting identification which is similar to HAR. Thus, rewriting identification not only reduces normal fragmentation but also provides the historical information for delta selection to save extra storage overhead.

RepEC-Core, the central module, includes two mechanisms, *delta-utilization-aware filter* and *cooperative cache scheme*. Delta-utilization-aware filter mainly focuses on identifying which containers need to be replicated and if so replicates them to proper nodes, implementing the replication side of RepEC-Core. The CCS, which enables RepEC-Core to be aware of both decoding recovery cache and regular restore cache, is used in the restore phase. It is designed to look up the required container, respectively, from the decoding cache and restore cache during restores. We will discuss more details of RepEC-Core in the next subsection.

3.2 | RepEC-HDS

To eliminate cyclic fragmentation in consecutive backups, we implement a History-aware Delta Selection scheme called RepEC-HDS. RepEC-HDS includes two components: rewriting identification and delta selection as shown in Figure 8.

Rewriting Identification. Rewriting identification includes *fragmented containers sequence* and *normal fragmentation elimination*. Fragmented containers sequence provides IDs of fragmented containers of the last backup for both *normal fragmentation elimination* and *delta selection*. When a duplicate chunk arrives, normal fragmentation elimination identifies if the container ID of this chunk exists in the fragmented containers sequence. Exist means that the chunk are fragmented and will be rewritten. Meanwhile, normal fragmentation elimination collects the information of fragmented containers of each backup.

Delta selection. Delta selection includes *base fragmentation sequence* and *cyclic fragmentation Identification*. Base fragmentation sequence records the fingerprints of base fragmentation of each backup. When a similar chunk arrives, cyclic fragmentation Identification first checks if its base chunk exists in the fragmented container sequence provided by rewriting identification. If exists, cyclic fragmentation identification further checks whether the base chunk of the similar chunk can be found in the base fragmentation sequence. If so, the base chunk of the

similar chunk is identified as cyclic fragmentation and the similar chunk will skip delta compression. Otherwise, delta selection performs delta compression between this chunk and its corresponding base chunk and update the fingerprints of its base chunk to the base fragmentation sequence. With the help of history information, we perform delta compression for noncyclic fragmentation, cyclic fragmentation can be greatly reduced.

3.3 | RepEC-Core

In this subsection, we will present an in-depth discuss on the two main components of the RepEC-Core with a focus on their data structures.

Delta-utilization-aware filter. The delta-utilization-aware filter module includes three data structures: *Replication Container Filter* (RCF), *Replication Node Selection*, and *Hybrid Reliability Scheme*. RCF defines several global variants: DCC number (Dnum for short) and non-DCC number (non-Dnum for short), and the delta utilization threshold (DUT). Different DUTs present different trade-off point between space overheads and restore performance (detailed in Section 4). Specifically, before writing chunks to the containers, RCF first checks whether the container is full or not. If not, we identify that the chunk is a DCC or a regular chunk. If it is a DCC chunk, we increase Dnum by 1, otherwise we increase non-Dnum by 1. Then the chunk will be directly written to the containers. If the container is full, we calculate its DU using Dnum/(Dnum + non-Dnum). If its DU exceeds DUT, the container needs to be replicated and thus will be sent to the the next structure *RNS*. Then we reset Dnum and non-Dnum for the next container.

After replication container filter, the qualified containers will be sent to the RNS. The main goal of RNS is to choose a suitable storage node for identified containers. Since we distribute the containers and parity objects to storage nodes in a round-robin fashion, we store containers' copies in the next node of the container own node. Here, we set the container position in its stripe to j. Assume we use an (k,m) erasure code configured with two parameters k and m ($m < k$) and the known stripeID, the container's node can be computed by the formula: nodeID = (j + stripeID) % (k+ m). Thus, the replicated container's node is (nodeID + 1). According to the computed result, we write the replicated containers to the corresponding nodes during the next structure *Hybrid Reliability Scheme*. When a DCC is lost, we can restore the chunk from its container copy, simultaneously keeping good cache locality. We describe the selection process of the replication node via an example shown in Figure 9. Figure 9 shows five files with six chunks each and we write them to six containers from C0 to C5 after deduplication and compression. Next, RCF computes the delta utilization of each container by using Dnum/(Dnum + non-Dnum) and the results are shown in Figure 9B. Assume that the DUT is 0.5, we find that containers C2 and C4 need to be replicated because their DUs are higher than 0.5 and other containers C0, C1, C3, and C5 will be operated on erasure coding. Recall that redundancy data is always placed behind data in reliable deduplication storage systems and then these data are distributed to storage nodes in a round-robin fashion.[38] We assume that a container's node is located is i, so this container's replication node is i+1. As shown in Figure 9C, the node of container C2 is 0 and the node of container C4 is 1. Correspondingly, their replicated container's nodes is 1 and 2, respectively.

HRS is designed to combine replication with erasure coding reliability approach based on different containers' characteristics. HRS creates a replicate structure and a hashtable R_hashtable to record the replication information, including the container's ID, the replicated container's ID and nodeID. After finding replicated containers and selecting the nodes, HRS first updates the replication information to R_hashtable and then uses

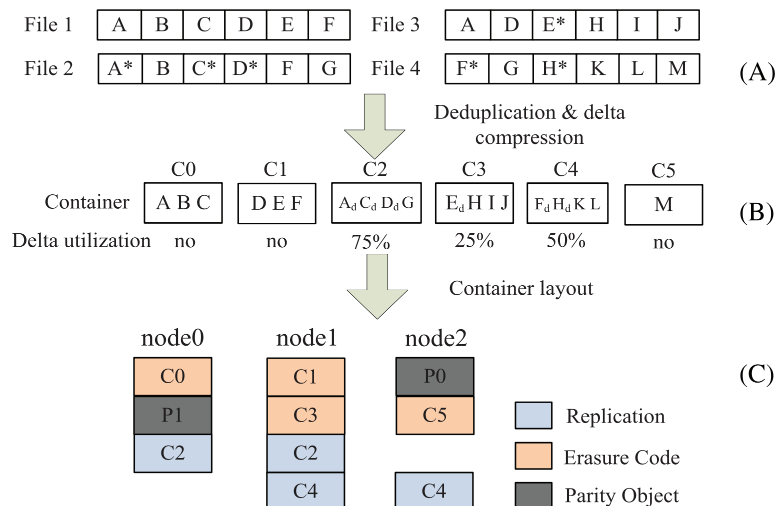

FIGURE 9 The selection process of the replication node

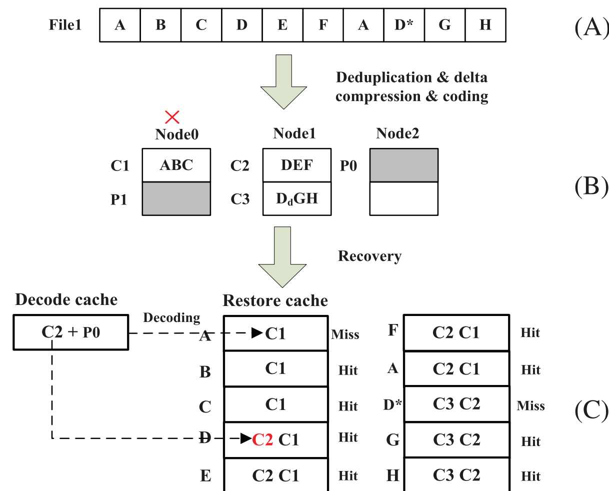

FIGURE 10 The principle of cooperative cache scheme

replication method to write the replicated containers to the selected node. Except for replicated containers, HRS implements Reed–Solomon coding as our erasure coding[39] to collect nonreplicated containers until the number of nonreplicated containers reaches *k*. *k* nonreplicated containers will be encoded to generate m parities.

To record the information about the replicated containers, we introduce additional metadata (i.e., replication information), as mentioned above. Assuming that metadata information is updated before their replicated containers are fully written, if nodes fail, an inconsistency problem will happen between the metadata information and their replicated containers. Specifically, during failure recovery, metadata about replicated containers will be found in the replication information, but the actual data will not, in fact, be stored in the nodes, resulting in ineffective recovery of their containers. Thus, how to ensure the consistency of additional metadata should be considered for our RepEC$^+$. We need to update the replication information only after the replicated containers have been completely written to their nodes. Specifically, RepEC$^+$ writes the replicated containers to the selected nodes after RCF and RNS. After the writing of the data to the replicated containers has been completed, the replication information begins to be updated in the hashtable R_hashtable. In doing this, although the replicated container has not yet been stored in the node, our RepEC$^+$ can keep the replicated containers synchronized to their replication information when the node fails to ensure data consistency. One other important concern, the overhead of this replication information, will be discussed in Section 4.

Cooperative cache scheme. The CCS module, which is only used for recovery, includes two data structures: *decoding cache structure* and *cooperative cache structure*. The main goal of CCS is to establish a relationship channel between a decoding cache and a restore cache to avoid container's extra read, thus enhancing the restore performance. Decoding cache structure creates a decoding cache to buffer the remaining available containers when a node fails. During restores, cooperative cache structure first checks whether a required container hits in the restore cache. If hits, the container is read from the restore cache. Otherwise, the restore cache looks up the container from the decoding cache. If the container exists, it is inserted into the restore cache and deleted from the decoding cache. Figure 10 pictorially illustrates the idea of CCS with an (2,1) erasure code. Assume that node0 fails, we read the container C2 and parity object P0 to decode the container C1. As shown in Figure 10C, container C2 and parity object P0 are buffered in the decoding cache and the decoded container C1 is stored in the restore cache with the LRU cache method.[4] Thus, chunks A, B, and C of File1 will be directly restored through the container C1. When we restore the chunk D in the container C2, we can find container C2 in the decoding cache instead of repeatedly reading from the node. Hence, container C2 is updated to the restore cache to provide the chunk D and remaining chunks, and then removed from the decoding cache.

4 | PERFORMANCE EVALUATION

We evaluate the performance of RepEC$^+$ in this section. Specifically, we first describe the experimental setup in Section 4.1. Next, in Section 4.2, we evaluate the reliability of RepEC$^+$ in terms of Mean Time To Data Loss (i.e., MTTDL). We find that the MTTDL of RepEC$^+$ becomes closer to replication scheme with the increase of the proportion of container replicas. In section 4.3, we evaluate the restore performance of RepEC$^+$. In addition, we conduct the sensitivity studies of restore performance through four aspects: DUT, cache allocation, container size and fragmentation

in Sections 4.4,4.5,4.6, and 4.7. In Section 4.8, we evaluate the write performance of RepEC$^+$. We observe that RepEC$^+$ has almost the same write performance as Reed–Solomon code, which means that write operation is not time consuming. Finally, we analyze the redundancy elimination ratio and storage overheads of our method, compared with the existing reliability schemes. We find that our RepEC$^+$ improves restore performance by 58.3%–76.7% with a low storage overhead and an acceptable redundancy elimination ratio.

4.1 | Experimental setup

Platform of the RepEC$^+$ **prototype.** We implemented a RepEC$^+$ prototype, including data deduplication, data compression, RepEC-HDS, and RepEC-Core for experimental evaluation. The server aggregates multiple storage nodes as a large storage pool, and provides the storage space for processed chunks. We also implemented HAR[23] for comparisons. We make comparisons among six schemes: Rep, RS, Random, RepEC, HAR* and RepEC$^+$. Table 2 gives the descriptions of six schemes. We use two libraries, including Jerasure 2.0[40] and GF-Complete[41] to implement Reed–Solomon coding scheme. In the circumstance of node failures, RepEC$^+$ implements the recovery of k data and parity containers from other available nodes to obtain failed containers. All evaluations are run on the Ubuntu 12.04.2 operating system which features a 16GB RAM, a quad core Intel i7-4770 processor at 3.4 GHz.

Configurations. RepEC$^+$ uses the Rabin-based chunking algorithm to divide a file to variable-size chunks[42] and sets an average chunk size of 8 kB. We choose the MD5 algorithm to compute chunks' fingerprints. We assume the fingerprint index pool is located in memory. We set a container size to be 4 MB. Xdelta,[43,44] are used for identifying similar data in a byte-wise sliding window. In this paper, we use a two-way mirroring replication and a (3,1) RS coding in our RepEC$^+$ prototype. Thus, the server aggregates four nodes as a single storage pool to store and manage data chunks.

Datasets description. Four datasets, including GCC, Linux, RDB, and Synthetic, are used in our evaluations and their characteristics are listed in Table 3. The Linux and GCC datasets are two representative backup workloads, which contain 250 Linux kernel release code files[45] and 88 backups of the GCC source code,[46] respectively. The RDB dataset has 100 backups of the redis key-value store database.[47] Besides above three datasets, we generate a large backup dataset called "Synthetic" by emulating real-world activity of file system,[48],[22] including "create," "delete," and "modify." The synthetic dataset has about 80% duplicates between adjacent backups. Since our paper focuses on restore performance when nodes fail, we mainly use the Synthetic dataset in the evaluation of restore performance.

Metrics. To make comparisons, we use four metrics, namely MTTDL, redundancy elimination ratio, write time and restore time, which are indicators of reliability, redundancy elimination efficiency, write performance, and restore performance, respectively. MTTDL is a traditional metric to characteristic the storage system reliability.[49] The redundancy elimination ratio is a key metric to measure a deduplicated and delta compressed storage systems.[9] In this paper, we define it as the percentage of redundant data eliminated by chunk-level deduplication and delta compression. Less restore time means better restore performance.

TABLE 2 The description of six reliability schemes

Scheme name	Description
Rep	Replication scheme
RS	Reed–Solomon coding scheme
Random	Randomly placing DCCs scheme
RepEC	RepEC$^+$ without RepEC-HDS
HAR*	RepEC-Core with HAR
RepEC$^+$	The system RepEC$^+$

TABLE 3 Characteristic of four datasets. DR represents deduplication ratio

Dataset name	Total size	Versions	DR	Redundancy elimination ratio
Linux	112GB	250	40%	95%
GCC	28GB	88	38%	80%
RDB	214GB	100	50%	88%
Synthetic	1436GB	500	84%	94%

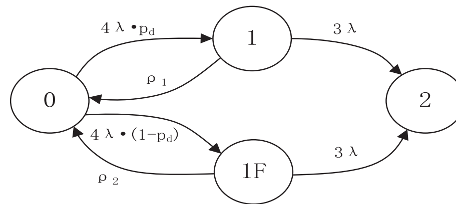

FIGURE 11 Markov reliability model for the RepEC+

TABLE 4 Reliability of Rep1/2, RS code and RepEC+

	MTTDL (years)
Rep1/2	1.19E+05
RS code	3.97E+04
RepEC+, P_d=0.5, C = 2	6.01E+04
RepEC+, P_d=0.6, C = 1.8	7.02E+04
RepEC+, P_d=0.7, C = 1.6	7.50E+04

Abbreviation: MTTDL, mean time to data loss.

4.2 | MTTDL analysis

Reliability is very important for the distributed storage systems[50] and can be characterized as MTTDL of the system.[51] The Markov model as a system model has been widely used to analyze the MTTDL. We extend a generalized Markov model to capture the special state transition of our RepEC+.

Modeling of RepEC+**.** To better demonstrate the reliability of RepEC+, we use a hybrid scheme with two-way mirroring replication and an (3,1) erasure coding. Figure 11 shows the Markov model diagram of RepEC+ method. In the diagram, the Markov model indicates the rates of container failures or system recoveries according to the state transition. The symbol in each state of the Markov chain denotes the number of failure containers. Since RepEC+ combines the replication with the erasure code to ensure the reliability, *State 0* can transit two states with four health containers. We use λ to describe the rate at which a container failure event occurs. Then, the transition rate from four containers healthy State 0 to one container failure State 1 is 4 λ. *State 1* denotes a state where one failure is decodable using replication. *State 1F* represents a state where one failure is decodable using erasure code. Let P_d denote the percentage of decodable one failure case using replication. Thus, the transition rate from *State 0* to *State 1* is $4\lambda p_d$ while the transition rate from *State 0* to *State 1F* is $4\lambda(1 - p_d)$.

In the node recovery phase, we assume that ρ_1 is recovery rate for *State 1* to *State 0*, which equals to the recovery rate of a single container failure. We assume each node with *S* storage capacity and ω available network bandwidth. We also use *N* to denote the total number of storage nodes. Thus, the number of nodes required in a single repair process is (*N*-1). Since the symbol *C* is used to denote the single repair cost. Thereby, the repair rate of a container failure is $\rho_1 = \rho_2 = \omega(N\text{-}1)/SC$.

Comparison. We set *N*, *S*, ω, $1/\lambda$ to 400, 16TB, 1Gbps,[49] 4 years,[52] respectively, and use them to evaluate the reliability of RepEC+. Table 4 shows the reliability of RepEC+, compared with two-way mirroring replication (Rep1/2) and Reed-Solomon (RS) code. In the table, it clearly shows that Rep1/2 achieves the better reliability than RS code and RepEC+. This is because three schemes have the same tolerance, the better recovery rate demonstrates the better reliability. The reliability of RepEC+ has a declination but still can be acceptable. Moreover, with the increase of the proportion of container replicas, we find that the reliability of RepEC+ becomes closer to Rep1/2 because replication can improve the reliability with the same level of tolerance.

4.3 | Restore performance

The restore time is a key metric of a deduplicated and delta-compressed storage system.[22] In this paper, using restore time, we evaluate the restore performance of RepEC+ through four aspects, namely DUT, cache allocation, container size and fragmentation. Figure 12 shows the restore performance comparisons of different five schemes. In our evaluations of Linux, RepEC+ outperforms Rep1/2, RS(3,1), Random and RepEC by 73%, 84%, 89%, and 76%, respectively. The reason is that RepEC+ not only addresses the inefficiency of cache but also reduces cyclic fragmentation. Since

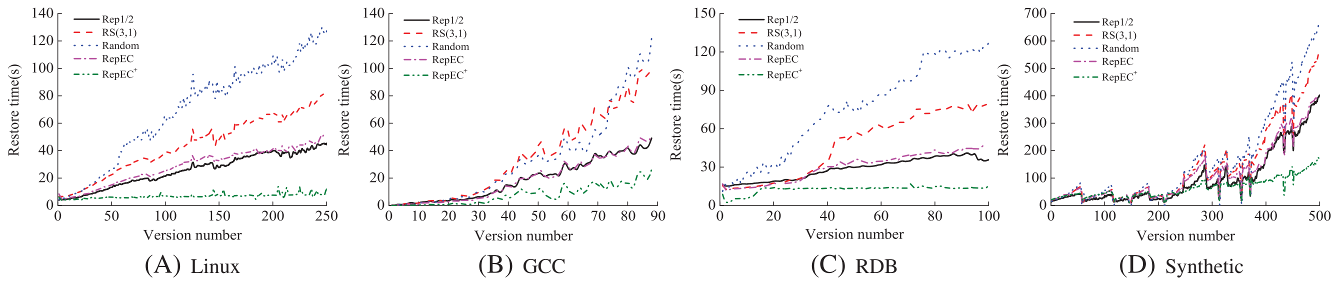

FIGURE 12 The restore performance of five reliability schemes on the four datasets. The restore cache/the decoding cache is 60 MB/4 MB, 56 MB/8 MB, 120 MB/8 MB, 240 MB/16 MB, respectively, in (A) Linux, (B) GCC, (C) RDB , and (D) Synthetic

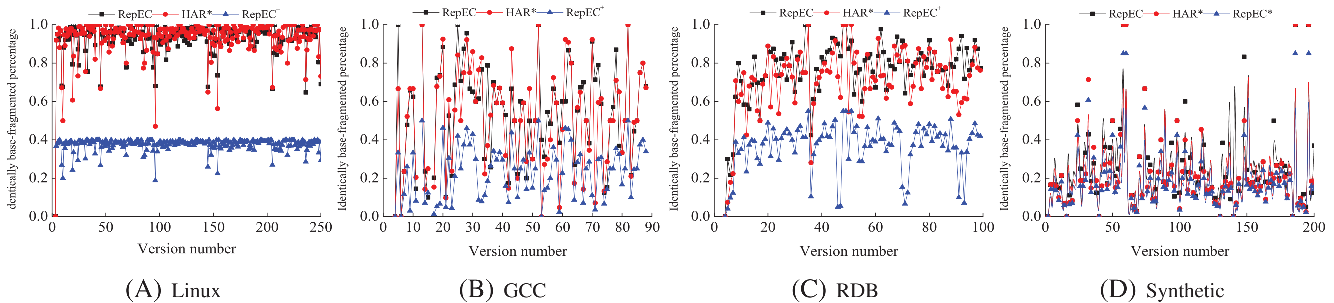

FIGURE 13 The percentage of base chunks of three reliability schemes on the four datasets. (A) Linux, (B) GCC, (C) RDB , and (D) Synthetic

the random placement of DCCs replicas destroys cache locality as describe in Section 2, Random has the worst restore performance among all the schemes. It demonstrates that RepEC$^+$ keeps cache locality by replicating a whole container based on the DU. Note that RepEC$^+$ is better than RepEC since it takes advantage of RepEC-HDS, which will be elaborated in Section 4.7. Moreover, the restore performance of RepEC is approximate to that of Rep1/2 in Linux. This is mainly because a CCS helps a restore cache to directly obtain the required container from a decoding cache to reduce the number of container reads.

The results in GCC, RDB, and Synthetic are similar with those in Linux. In GCC, RDB, and Synthetic, RepEC$^+$ achieves best restore performance among five schemes. For example, in Synthetic, RepEC$^+$ outperforms Rep1/2, RS(3,1), Random and RepEC by 34%, 54%, 61%, and 44%, respectively. The reasons are twofold. First, RepEC$^+$ can directly obtain a container from its copy, rather than reading three containers from a stripe. Second, compared with other four schemes, RepEC$^+$ prevents cyclic fragmentation to improve the utilization of container, which shortens the restore time, specially the decoding time. In addition, we observe that the restore performance of RepEC is sightly better than that of Rep1/2 in a few versions of the GCC dataset. This is because the containers from the same stripe probably belong to multiple files in deduplicated and delta compressed storage systems. Thus, the design of CCS not only avoids container being read repeatedly but also increases the chance of container hits.

As observed in Section 2.4, there exists cyclic fragmentation in consecutive backups. To illustrate that our method dramatically eliminates the cyclic fragmentation, we use identically base-fragmented percentage to evaluate the RepEC, HAR*, RepEC$^+$ in Figure 13. Generally, RepEC$^+$ greatly reduces the cyclic fragmentation. For example, the identically base-fragmented percentage of RepEC$^+$ is about 60%, 50%, 45%, and 11% lower than that of RepEC in Linux, GCC, RDB, and Synthetic datasets, respectively. This is because base fragmentation will not be fragmented in consecutive backups, which means that more useful chunks are stored in the containers. Containers without cyclic fragmentation contribute to the improvement of restore performance and thus RepEC$^+$ has the best restore performance among all the schemes as shown in Figure 12. Furthermore, we find that the identically base-fragmented percentage of HAR* and RepEC curves are nearly overlapped. It means that HAR method has no ability to eliminate the cyclic fragmentation because it only aims to alleviate the normal fragmentation, as discussed previously in Section 2.4. In summary, our method efficiently reduces the cyclic fragmentation and promotes the restore performance improvement further.

4.4 | Impact of DUTs on restore performance

DUT determines the number of replicated containers. Impacts of varying the DUT on restore performance is shown in Figure 14. With the increase of DUTs, the restore time gradually increases. It illustrates that low DUT can find more containers to be replicated. The more containers' copies

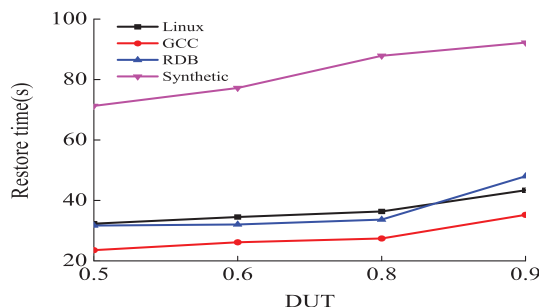

FIGURE 14 Impacts of varying the delta utilization threshold on restore performance. The cache size is 64, 64, 128, and 256 MB in Linux, GCC, RDB, and Synthetic, respectively

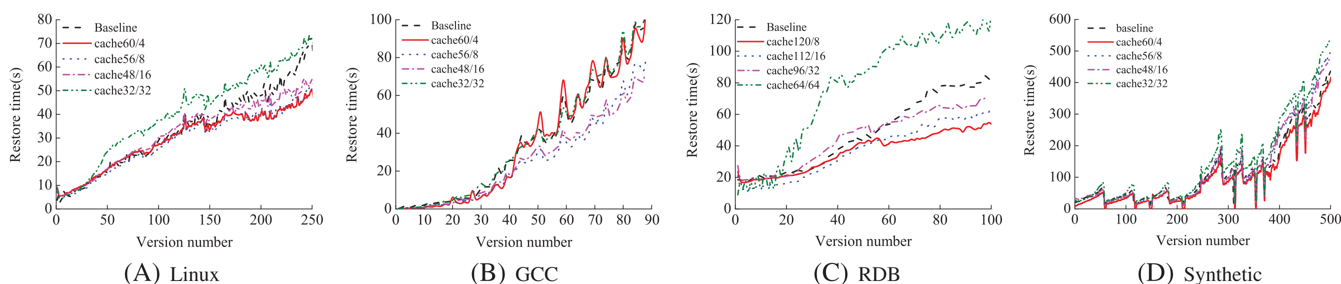

FIGURE 15 Impacts of cache allocations on restore performance. delta utilization threshold is 0.9. Baseline presents the RepEC without CCS. The numbers of each legend represent the size of restore cache size (the number above the slash) and the size of decoding cache size (the number under the slash) in terms of MB. (A) Linux, (B) GCC, (C) RDB , and (D) Synthetic

will improve the restore performance, but correspondingly they occupy more storage space. Thus, the DUT introduces a tradeoff between restore performance and space overhead. Note that the restore performance of RDB between 0.5 and 0.8 shows a smooth trend, compared to the GCC dataset because of different datasets' characteristics. It means that the DU of each container is about 0.8 in the Linux and RDB datasets. Likewise, the restore performance of Synthetic degrades gradually and becomes stable until DU reaches 0.8. It indicates that the DU of each container is mainly between 0.5 and 0.6 in Synthetic. In GCC, the DU of each container distributes between 0.5 and 0.9 and thus rises continuously.

4.5 | Impact of cache allocations on restore performance

As discussed in Section 2.3, Figure 15 shows the impact of cache allocations on restore performance to demonstrate the efficiency of CCS. On the Linux dataset, the restore performance of cache 60/4 and 56/8 outperform the baseline by 13% and 15%, respectively. This suggests that the design of CCS further enhances the restore performance and efficiently solves the challenge 2. However, with the increase of the decoding cache, the restore time increases. For example, cache 32/32 has the worst restore performance than others. This is because the Linux dataset suffers from many fragmentation and a large restore cache can alleviate the fragmentation.[23] Thus, with the increase of version numbers, the restore performance of cache 60/4 enjoys more benefits due to the large restore cache, such as 60 MB. Similarly, the RDB and Synthetic datasets have nearly the same trend like Linux.

Different from other three datasets, cache 56/8 and cache 48/16 achieve better restore performance than others in GCC. This is because that GCC does not have as much redundancy as Linux, which has a small impact on the restore cache. Conversely, a relatively large decoding cache is able to hold more containers used immediately.

4.6 | Impact of container sizes on restore performance

We evaluate the impact of container sizes on restore performance in this section. The experimental results are shown in Figure 16. It can be observed that, with the increase of the container sizes, the restore performance gradually degrades. Clearly, our solution has the best restore performance under the 4 MB container size in the four datasets. This result implies that a large container has a negative impact on the restore performance for

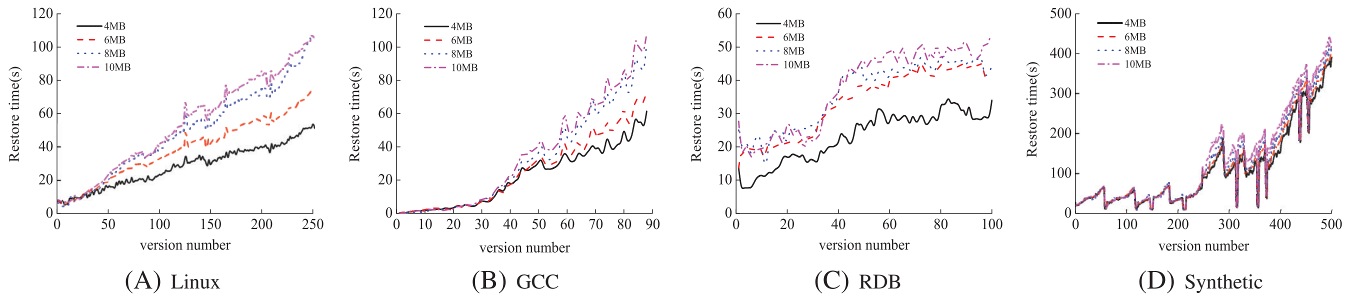

FIGURE 16 Impacts of container sizes on restore performance. The container size varies 4, 6, 8, and 10MB. (A) Linux, (B) GCC, (C) RDB , and (D) Synthetic

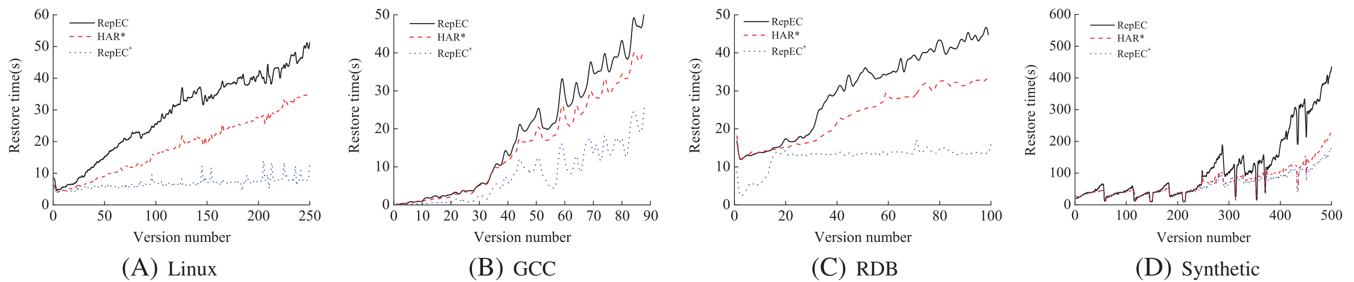

FIGURE 17 Impacts of fragmentation on restore performance. (A) Linux, (B) GCC, (C) RDB , and (D) Synthetic

all the datasets. The main reason is that, compared with a small container, the large container will hold more ineffective chunks, namely fragmented data.[23] The recovery of fragmented data will cause more containers to be read during both erasure coding failure recovery and file restore. Therefore, a small container can alleviate the problem of reading containers many times in the deduplicated and delta-compressed storage systems. In this paper, we chose 4 MB as the container size to evaluate the experiments.

4.7 | Impact of fragmentation on the restore performance

Cyclic fragmentation will degrade restore performance as discussed in Section 2.4, we evaluate the the impact of fragmentation on restore performance of RepEC, HAR* and RepEC$^+$. Figure 17 shows the experimental results. In Linux, GCC, RDB, and Synthetic, RepEC$^+$ further improves the restore performance than RepEC by 76.7%, 58.4%, 58.3%, and 44.8%, respectively. This is because RepEC-HDS selectively performs delta compression between noncyclic fragmentation to reduce cyclic fragmentation. Meanwhile, RepEC$^+$ further improves the restore performance than HAR* by 64.1%, 48.8%, 46.8%, and 11.3%. These results imply that the design of RepEC-HDS eliminates not only normal fragmentation but also base fragmentation compare to HAR*. Note that RepEC$^+$ performs slightly better than HAR* in Synthetic. Since Synthetic has less redundancy after deduplication compared to other three datasets, less base fragmentation arises during delta compression. In addition, compare to RepEC and HAR*, the restore performance of RepEC$^+$ will not increase by a wide margin when it reaches a value. Like Linux and RDB, as the restored version increases, the restore performance of RepEC$^+$ keeps stable. It is mainly because most of cyclic fragmentation are nearly same in consecutive backups, each backup almost reduces the same number of cyclic fragmentation.

4.8 | Write performance

Write time is an important metric during backups, so we use it to evaluate the write performance of our RepEC$^+$. The write time of four schemes is shown in Figure 18. In this experiment, we set the DUT for the Linux, GCC, and RDB datasets to 0.9, 0.6, and 0.8, respectively. In theory, RS(3,1) should obtain the best write performance due to writing the least number of containers among all the schemes. Nevertheless, RepEC$^+$ has almost the same write performance as RS(3,1) in both the Linux and RDB datasets, as can be seen from Figure 18A,C. Recall that the DUT is used to limit how many containers will be written to the nodes. The larger the DUT is, the fewer replicated containers the nodes will have. When the DUT is

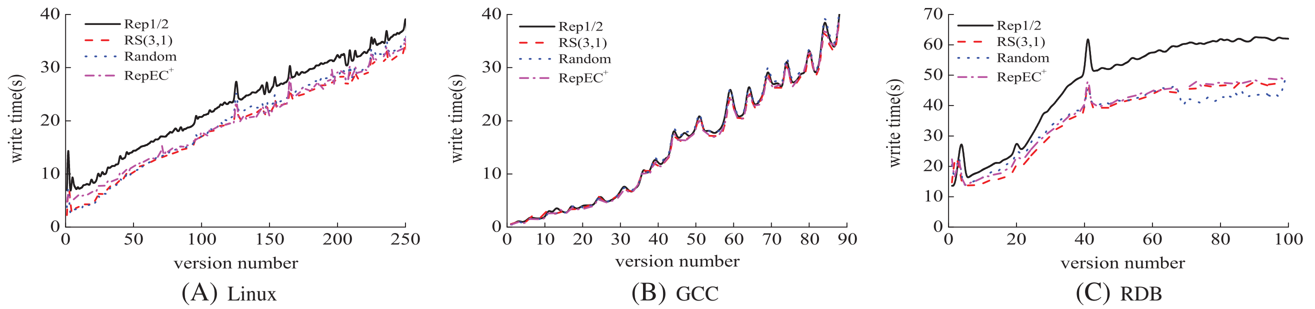

FIGURE 18 The write performance of four schemes on three datasets. (A) Linux, (B) GCC, and (C) RDB ,

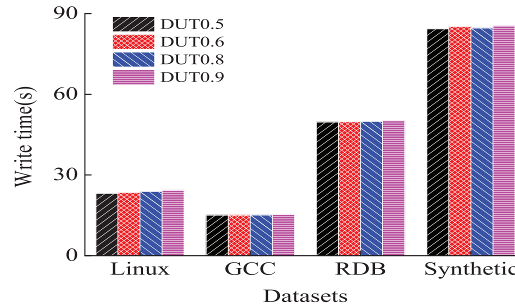

FIGURE 19 The write performance of the RepEC+ with different delta utilization thresholds. The cache size is 64, 64, 128, and 256 MB in Linux, GCC, RDB, and Synthetic, respectively

0.9 and 0.8 in the Linux and RDB datasets, only a few replicated containers will be stored in the nodes. This suggests that the number of written replicated containers in RepEC+ is close to that in RS(3,1) (e.g., 1325 vs. 1300 in the Linux dataset).

Different from the Linux and RDB datasets, all the schemes in the GCC dataset have similar write performance, as can be seen in Figure 18B. This is because each file in the GCC dataset is small, which means that it does not need many replicated containers to be written. Thus, this write operation is not time consuming. In other words, this implies that the process of writing the containers to the nodes is not the bottleneck for small datasets. Note that the write performance of Rep1/2 is the worst among all the schemes for the Linux, GCC, and RDB datasets. Since all the containers in Rep1/2 need to be stored twice in the nodes, Rep1/2 writes more containers than the other three schemes and therefore gets the worst write performance.

In addition, RepEC+ does not vary greatly in write performance under different DUTs as shown in Figure 19. Recall that the restore performance of RepEC+ in 0.9 is worse than that in 0.5, 0.6, and 0.8 for the four datasets in Figure 14. However, we find that write performance remains stable as DUT increases. Compared with other phases, write operations, such as written replicated containers will not take for a long time as above analysis. In other words, the increase in overhead for writing containers can be ignored for different workloads. Thus, the write time with different DUTs only impact both storage overheads and restore performance but not write performance.

4.9 | Redundancy elimination ratio

In order to ensure the reliability of deduplicated and delta compressed storage systems, reliability schemes will introduce redundancies and thus have an impact on the redundancy elimination ratio. To illustrate the impact of reliability schemes on redundancy elimination ratios, Figure 20 shows the redundancy elimination ratio with or without reliability schemes on the four datasets. Baseline represents the system without any reliability schemes. For the Linux dataset, the redundancy elimination ratios of baseline, Rep1/2, RS(3,1), Random, RepEC are 0.95, 0.82, 0.89, 0.88 and 0.87 respectively. These results imply that redundancies introduced by reliability schemes decrease the redundancy elimination ratios. Note that the redundancy elimination ratio of Rep1/2 is worst because Rep1/2 has the most redundancy overheads among all the reliability schemes. Although the redundancy elimination ratio of our solution has a slight decrease about 4%–9%, it can be acceptable because our scheme achieves restore performance improvement of 58%–76% on the four datasets.

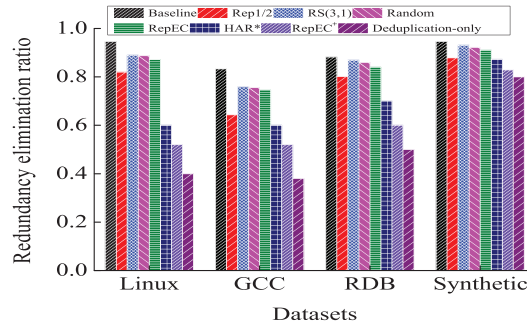

FIGURE 20 The redundancy elimination ratio of different reliability schemes on the four datasets

TABLE 5 Space overhead for recovery. Re% represents the reduction%

Reliability scheme	Linux	GCC	RDB	Synthetic
Rep1/2	12.8 GB	10 GB	26.1 GB	100.5 GB
RS code	4.24 GB	3.35 GB	8.71 GB	27.5 GB
RepEC$^+$0.5/Re%	5.4 GB/54%	3.9 GB/61%	12 GB/54%	40.24 GB/59%
RepEC$^+$0.7/Re%	5.3 GB/57%	2.12 GB/79%	12 GB/54%	13.55 GB/86%
RepEC$^+$0.8/Re%	5.2 GB/58%	1.3 GB/87%	11.5 GB/56%	9.12 GB/89%
RepEC$^+$0.9/Re%	3.9 GB/70%	0.2 GB/98%	1.6 GB/94%	9.03 GB/91%

In addition, Figure 20 also shows the impact of RepEC-HDS on the redundancy elimination ratio. Note that the redundancy elimination ratio of RepEC, HAR*, and RepEC$^+$ is 0.8, 0.6, and 0.5, respectively, in GCC dataset. It shows that HAR* rewrites more data but improves the restore performance as shown in Figure 17. The redundancy elimination ratio of RepEC$^+$ is close to that of HAR*, which means that RepEC-HDS does not sacrifice the redundancy elimination ratios to eliminate the existence of cyclic fragmentation. This is because base fragmentation that exist in fragmented containers only account for 0.1% of all the base chunks. Moreover, We also include the results of deduplication-only, which means applying delta compression over deduplication further significantly removes content redundancy.

4.10 | Storage overheads analysis

Table 5 shows the comparisons between RepEC$^+$ and other reliability schemes in terms of storage overhead on the four datasets. Recall that a small DUT presents more containers are replicated and thus takes more space. We observe that even in the case of DUT being 0.5, RepEC$^+$ still saves 54%, 61%, 54%, and 59% more storage storage than Rep1/2, respectively, in the Linux, GCC, RDB, and Synthetic datasets. This reason is that RepEC$^+$ makes replication for eligible containers according to the DUT and uses erasure codes with less storage overhead for noneligible containers. Take the Linux dataset for example. We need to provide the reliability for 2500 containers. Using Rep1/2, it introduces redundancies about 2500 containers because of retaining replications for all the containers. Using RepEC$^+$, it generates redundancies about $950 + 1550/3 \approx 1467$ containers since RepEC$^+$ makes replication for 950 containers and a (3,1) RS code for 1550 containers when DUT is 0.5. Thus, RepEC$^+$ greatly saves storage space compared to Rep1/2. Meanwhile, space overhead is highly related to the DUT. Note that, with the increase of the DUT, RepEC$^+$ needs less replication storage space. This is because the containers to be replicated becomes less when the DUT increases, so replication storage space also becomes less. Specifically, when the DUT is 0.9, 0.7, 0.8, RepEC$^+$ saves storage space by 70%, 79%, and 56% on the Linux, GCC, and RDB datasets. As the restore performance concerns, RepEC$^+$ outperforms RS code by 35%, 47%, and 26%, even though it sacrifices storage overhead compared to RS code. Overall, our approach RepEC$^+$ not only significantly increases the restore performance but also achieves large space saving. In addition, combined with Figure 14 and Table 5, we also observe the characteristic of the DCC distribution in container. When the DU is lower than 0.9, the storage space nearly incurs no changes or less changes in Linux and RDB. For example, RepEC$^+$0.5 and RepEC$^+$0.7 save storage space by 54% and 56%, respectively, on the RDB dataset. It proves that most of DUs keep 0.8 and the distribution of redundant data is relatively uniform. For the Synthetic dataset, the DU of is lower than 0.7 as there is a little changes of storage overhead among 0.7, 0.8, and 0.9. In this range, the restore performance of Synthetic dataset keeps smoothly as shown in Figure 14.

Moreover, in our method, we analyze the overhead of the replication information introduced by the replicated containers. Because the replication information consists of the container's ID, the replicated container's ID and the nodeID, the storage overhead of a replication structure in RepEC$^+$ is 4 B + 4 B + 4 B = 12 B. That is, the extra overhead caused by a replicated container in RepEC$^+$ is 12 B. From the Table 5, we know that the Linux dataset will generate 5.4 GB/4 MB = 1000 MB containers, so the maximal cost of its replicated containers is 12 B*1000 = 10 MB. Similarly, the GCC dataset needs (0.2 GB/4 MB)*10 B = 1 MB additional replication overhead. Thus, RepEC$^+$ adds the least storage cost, accounting for 0.2% and 0.03% of the storage overhead in Linux and GCC, respectively. In summary, compared with its large storage overhead, overhead associated with the replicated information is negligible in RepEC$^+$.

Finally, we analyze the storage overhead of the decoding cache structure and cooperative cache structure. Since the decoding cache structure buffers the remaining available containers when nodes fail and records the cache size, the storage overhead of the decoding cache structure is 4B*Tnum+4B. *Tnum* represents that the total number of containers that can be stored in the decoding cache. As discussed in Section 4.5, the cache 60/4 achieves better restore performance than other cache allocation schemes. Take Synthetic representing a large workload with about 3 GB each version for example. Assume that the total cache size is 80% of each version, the decoding cache size is 3 GB*80%*4/60 ≈ 164 MB and *Tnum* is 164 MB/4 MB(one container size) = 41 containers. Thus, the storage overhead of the decoding cache structure consumes 4 B*41+4 B≈ 0.2 kB. Likewise, the cooperative cache structure records a hit count and a miss count, and thus its storage overhead is 4 B + 4 B = 8 B. In summary, the storage overhead of the decoding cache and cooperative cache will incur storage overhead but are within an acceptable range.

5 | CONCLUSION

Motivated by our observation that DCCs are far smaller than regular chunks, we propose RepEC$^+$, a hybrid reliability scheme in deduplicated and delta-compressed storage systems to ensure reliability while improving restore performance. A delta-utilization-aware filter in RepEC$^+$ is designed to adaptively use replication scheme to containers based on delta utilization to improve cache locality. Meanwhile, a cooperative cache in RepEC$^+$ merges recovery cache and restore cache to greatly reduce containers' read, thus achieving the nearly same restore performance as replication. In addition, we implement a History-aware Delta Selection (RepEC-HDS) that takes advantage of historical information to identify and reduce cyclic fragmentation. With the help of RepEC-HDS, RepEC$^+$ significantly improves the restore performance by 58.3%–76.7% with a low storage overhead.

ACKNOWLEDGMENTS

This work is supported in part by NSFC in grant numbers 61832020 and 61772216, National Key R&D Program of China in grant number 2018YFB10033005, National Defense Preliminary Research Project (31511010202), National Science and Technology Major Project in grant number 2017ZX01032-101, Fundamental Research Funds for the Central Universities.

DATA AVAILABILITY STATEMENT

The data that support the findings of this study are openly available. The Linux and GCC datasets are publicly available at [https://www.kernel.org/][45] and [http://ftp.gnu.org/gnu/][46] respectively. The RDB dataset is collected from [http://redis.io/].[47] Further, the Synthetic dataset is available from the corresponding author upon reasonable request.

ORCID

Chunxue Zuo https://orcid.org/0000-0002-2374-5772

REFERENCES

1. Gantz J, Reinsel D. The digitization of the world from edge to core. DATA AGE 2025; 2018.
2. Meyer DT, Bolosky WJ. A study of practical deduplication. *ACM Trans Storage (TOS)*. 2012;7(4):14:1-14:20. doi:10.1145/2078861.2078864
3. Kulkarni P, Douglis F, LaVoie JD, Tracey JM. Redundancy elimination within large collections of files. Proceedings of USENIX Annual Technical Conference; 2004:59-72; Boston, MA.
4. Zhu B, Li K, Patterson RH. *Avoiding the Disk Bottleneck in the Data Domain Deduplication File System FAST'08*. USENIX Association; 2008:269-282.
5. Zhang Y, Jiang H, Feng D, et al. AE: an asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication. Proceedings of IEEE Conference on Computer Communications; 2015:1337-1345; Kowloon, Hong Kong. 10.1109/INFOCOM.2015.7218510
6. Xia W, Jiang H, Feng D, et al. A comprehensive study of the past, present, and future of data deduplication. *Proc IEEE*. 2016;104(9):1681-1710. doi:10.1109/JPROC.2016.2571298
7. Guo F, Efstathopoulos P. Building a high-performance deduplication system. Proceedings of USENIX Annual Technical Conference; 2011.
8. Xia W, Jiang H, Feng D, Tian L. Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets. Proceedings of the Data Compression Conference;2014; 2014:203-212; Snowbird, UT. 10.1109/DCC.2014.38
9. Zhang Y, Feng D, Hua Y, et al. Reducing chunk fragmentation for in-line delta compressed and deduplicated backup systems; 2017:1-10; IEEE NAS. Shenzhen, China. 10.1109/NAS.2017.8026874

10. Muthitacharoen A, Chen B, Mazieres D. A low-bandwidth network file system. Proceedings of the 18th ACM Symposium on Operating System Principles; 2001:174-187; Alberta, Canada. 10.1145/502034.502052
11. Shilane P, Huang M, Wallace G, Hsu W. WAN-optimized replication of backup datasets using stream-informed delta compression. *ACM Trans Storage*. 2012;8(4):13:1-13:26.
12. Yu H, Xue L, Dan F. Neptune: efficient remote communication services for cloud backups. Proceedings of the IEEE Conference on Computer Communications (INFOCOM); 2014:844-852; Toronto, Canada. 10.1109/INFOCOM.2014.6848012
13. Yong C, Lai Z, Xin W, Dai N. QuickSync: improving synchronization efficiency for mobile cloud storage services. *IEEE Trans Mob Comput*. 2017;16(12):3513-3526.
14. Dropbox. https://www.dropbox.com
15. Shilane P, Wallace G, Huang M, Hsu W. Delta compressed and deduplicated storage using stream-informed locality. Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems; 2012; Boston, MA.
16. Zuo C, Wang F, Huang P, Hu Y, Feng D. RepEC-duet: ensure high reliability and performance for deduplicated and delta-compressed storage systems. Proceedings of IEEE 37th International Conference on Computer Design (ICCD); 2019:190-198; Abu Dhabi, United Arab Emirates. 10.1109/ICCD46524.2019.00032
17. Bhagwat D, Pollack KT, Long DDE, Schwarz TJE, Miller EL, Paris JF. Providing high reliability in a minimum redundancy archival storage system. Proceedings of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS); 2006:413-421; Monterey, CA. 10.1109/MASCOTS.2006.42
18. Liu C, Yu G, Sun L, Yan B, Wang D. R-ADMAD: high reliability provision for large-scale de-duplication archival storage systems. Proceedings of the 23rd International Conference on Supercomputing; 2009:370-379; New York, ACM. 10.1145/1542275.1542327
19. Dubnicki C, Gryz L, Heldt L, et al. HYDRAstor: a scalable secondary storage. Proceedings of the 7th USENIX Conference on File and Storage Technologies; 2009:197-210; San Francisco, CA.
20. Li X, Lillibridge M, Uysal M. Reliability analysis of deduplicated and erasure-coded storage. *SIGMETRICS Perform Eval Rev*. 2010;38(3):4-9.
21. Kaczmarczyk M, Barczynski M, Kilian W, Dubnicki C. Reducing impact of data fragmentation caused by in-line deduplication. Proceedings of the 5th Annual International Systems and Storage Conference; 2012:11; Haifa, Israel. 10.1145/2367589.2367600
22. Lillibridge M, Eshghi K, Bhagwat D. Improving restore speed for backup systems that use inline chunk-based deduplication. Proceedings of the 11th USENIX Conference on File and Storage Technologies; 2013:183-198; San Jose, CA.
23. Fu M, Feng D, Hua Y, et al. Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information. Proceedings of USENIX Annual Technical Conference; 2014:181-192; Philadelphia, PA.
24. Zhang P, Ping H, He X, Hua W, Yan L, Ke Z. RMD: a resemblance and mergence based approach for high performance deduplication. Proceedings of the International Conference on Parallel Processing; 2016:536-541; Philadelphia, PA. 10.1109/ICPP.2016.68
25. Zuo C, Wang F, Huang P, Hu Y, Feng D, Zhang Y. PFCG: improving the restore performance of package datasets in deduplication systems. Proceedings of IEEE 36th International Conference on Computer Design (ICCD); 2018:553-560; Orlando, FL. 10.1109/ICCD.2018.00088
26. Meister D, Kaiser J, Brinkmann A. A study on data deduplication in HPC storage systems. Proceedings of Conference on High Performance Computing Networking, Storage and Analysis; 2012:7.
27. Wallace G, Douglis F & Qian H et al. Characteristics of backup workloads in production systems. Proceedings of the 10th USENIX Conference on File and Storage Technologies; 2012:4; Santa Clara, CA.
28. Zhen S, Kuenning G, Mandal S, et al. A long-term user-centric analysis of deduplication patterns. Proceedings of 32nd Symposium on Mass Storage Systems and Technologies; 2016:1-7. 10.1109/MSST.2016.7897080
29. Douglis F, Iyengar A. Application-specific delta-encoding via resemblance detection. Proceedings of the Usenix Technical Conference; 2003:113-126.
30. Xia W, Jiang H, Feng D, Tian L, Fu M, Zhou Y. Ddelta: a deduplication-inspired fast delta compression approach. *Perform Eval*. 2014;79:258-272.
31. Wu S, Li KC, Mao B, Liao M. DAC: improving storage availability with deduplication-assisted cloud-of-clouds. *Future Gener Comput Syst*. 2016;74:190–198.
32. Xiao M, Hassan MA, Xiao W, Qi W, Chen S. CodePlugin: plugging deduplication into erasure coding for cloud storage. Proceedings of the Usenix Conference on Hot Topics in Cloud Computing; 2015; Santa Clara, CA.
33. Wu S, Luan H, Mao B, Jiang H, Zhou J. Improving reliability of deduplication-based storage systems with per-file parity. Proceedings of the IEEE 37th Symposium on Reliable Distributed Systems (SRDS); 2018:171-180; Salvador, Brazil. 10.1109/SRDS.2018.00028
34. Rozier EWD, Sanders WH. A framework for efficient evaluation of the fault tolerance of deduplicated storage systems. Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks; 2012:1-12; Boston, MA. 10.1109/DSN.2012.6263921
35. Fu M, Han S, Lee PPC, Feng D, Chen Z, Yu X. A simulation analysis of redundancy and reliability in primary storage deduplication. *IEEE Trans Comput*. 2018;67:1259-1272.
36. Srinivasan K, Bisson T, Goodson G, Voruganti K. iDedup: latency-aware, inline data deduplication for primary storage. Proceedings of the 10th USENIX conference on File and Storage Technologies; 2012:24.
37. Ng CH, Lee PP. Revdedup: a reverse deduplication storage system optimized for reads to latest backups. Proceedings of the Asia-Pacific Workshop on Systems; 2013:15:1-15:7; Singapore. 10.1145/2500727.2500731
38. Plank JS, Luo J, Schuman CD, Xu L, Wilcox-O'Hearn Z. A performance evaluation and examination of open-source erasure coding libraries for storage. Proceedings of the 7th USENIX Conference on File and Storage Technologies; 2009; San Francisco, CA.
39. Reed IS, Solomon G. Polynomial codes over certain finite fields. *J Soc Ind Appl Math*. 1960;8(2):300-304.
40. Plank JS, Greenan KM. Jerasure: a library in C facilitating erasure coding for storage applications C version 2.0. Technical Report UT-EECS-14-721, University of Tennessee; 2014.
41. Plank JS, Miller EL, Greenan KM, et al. GF-complete: a comprehensive open source library for Galois field arithmetic version 1.0. Technical report UT-CS-13-716, University of Tennessee; 2013.
42. Rabin MO. *Fingerprinting by Random Polynomials*. Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University; 1981.
43. Macdonald JP. *File System Support for Delta Compression*. Lecture Notes in Computer Science. Department of Electrical Engineering and Computer Science, University of California at Berkeley; 2000.

44. Trendafilov D, Memon N, Suel T. *zdelta: An Efficient Delta Compression Tool*. Department of Computer and Information Science, Polytechnic University Technical Report; 2002.
45. Linux archives; 2019. https://www.kernel.org/
46. Gnu archives; 2018. http://ftp.gnu.org/gnu/
47. Sanfilippo S, Noordhuis P. Redis key-value database; 2010. http://redis.io/
48. Tarasov V. Generating realistic datasets for deduplication analysis. Proceedings of the 2012 USENIX Conference on Annual Technical Conference; 2012:261-272; Boston, MA.
49. Cheng H, Simitci H, Xu Y, et al. Erasure coding in windows azure storage. Proceedings of the Usenix Conference on Technical Conference; 2012:15-26; Boston, MA.
50. Chun BG, Dabek F, Haeberlen A, et al. Efficient replica maintenance for distributed storage systems. Proceedings of the Conference on Networked Systems Design and Implementation; 2006; San Jose, CA. http://www.usenix.org/events/nsdi06/tech/chun.html.
51. Qin X, Miller EL, Thomas S, Long DDE, Brandt SA, Litwin W. Reliability mechanisms for very large storage systems. Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies; 2003:146-156.
52. Sathiamoorthy M, Asteris M, Papailiopoulos D, et al. XORing elephants: novel erasure codes for big data. *Proc VLDB Endow*. 2013;6(5):325-336.

How to cite this article: Zuo C, Wang F, Zheng M, Hu Y, Feng D. Ensuring high reliability and performance with low space overhead for deduplicated and delta-compressed storage systems. *Concurrency Computat Pract Exper*. 2021;e6706. doi: 10.1002/cpe.6706