Reference-counter Aware Deduplication in Erasure-coded Distributed Storage System

Tong Liu[1], Xubin He[1], Shakeel Alibhai[1], and Chentao Wu[2]

[1]Department of Computer and Information Sciences , Temple University
[2]Department of Computer Science and Engineering , Shanghai Jiao Tong University

***Abstract*—In modern distributed storage systems, space efficiency and system reliability are two major concerns. As a result, contemporary storage systems often employ data deduplication and erasure coding to reduce the storage overhead and provide fault tolerance, respectively. However, little work has been done to explore the relationship between these two techniques. In this paper, we propose Reference-counter Aware Deduplication (RAD), which employs the features of deduplication into erasure coding to improve garbage collection performance when deletion occurs. RAD wisely encodes the data according to the reference counter, which is provided by the deduplication level and thus reduces the encoding overhead when garbage collection is conducted. Further, since the reference counter also represents the reliability levels of the data chunks, we additionally made some effort to explore the trade-offs between storage overhead and reliability level among different erasure codes. The experiment results show that RAD can effectively improve the GC performance by up to 24.8% and the reliability analysis shows that, with certain data features, RAD can provide both better reliability and better storage efficiency compared to the traditional Round-Robin placement.**

I. Introduction

As the size of online digital content grows explosively every day, the space efficiency and data reliability of storage systems are attracting more and more attention to researchers. In recent years, some techniques have been proposed to address these two concerns. One such method, known as deduplication, is a well-developed technology that can improve the storage efficiency of a storage system by identifying and eliminating duplicate data chunks [1]–[3]. By adopting deduplication, a storage system can improve storage efficiency by up to 20x [4]. However, while deduplication improves space efficiency, it fundamentally changes the reliability of objects. Traditionally, storage systems such as HDFS [5], GFS [6], and Azure [7] employ triple replications to ensure data reliability and availability. However, after deduplication, only unique chunks are stored, and these are potentially spread across multiple disks and shared by various objects. As a result, replication no longer provides fault tolerance to the system. In this case, erasure coding can be considered. Erasure coding encodes the original data blocks by adding redundant data blocks, such that if a certain number of original data blocks fail, we can always reconstruct the failed data blocks with the encoded blocks. Erasure coding can improve storage efficiency with far less storage redundancy, while also achieving the same or even higher fault tolerance than replication [8], [9]. Several works have been proposed recently which integrate erasure coding into deduplication to ensure both high data reliability and high availability [10]–[13].

Although erasure coding has these attractive features, it also has its drawbacks. Erasure coding not only suffers from the low read/write performance when degraded I/O happens, but it also requires more computation resources since it needs to perform encoding for all new incoming data and decoding for reconstructing failed data. Further, in a real storage system, as many duplications are intended by users and applications for increased reliability or availability, after deduplication, the reliability levels of the data blocks actually vary with each other, which in some degree can be represented by its reference counter. (The value of the chunk's reference counter indicates how many times this chunk is shared among all the objects in the system.) As dependencies are introduced between files that share the same chunks, if a chunk with a high reference counter is lost, a large amount of data will become inaccessible due to the unavailability of all the files that share this chunk [14]. Therefore, simply employing erasure coding into deduplication and encoding the unique data chunks with no strategy may not be a good choice with respect to system performance and reliability.

In this paper, we propose RAD, Reference-counter Aware and erasure-coded Deduplication scheme. To our knowledge, little work has been done to discover the relationship between erasure coding and deduplication. In this work, we try to combine the features of these two techniques to optimize the storage system by improving garbage collection performance when a deletion operation occurs and explore the trade-offs between reliability, space efficiency, and system performance among different erasure codes in erasure-coded deduplication storage systems.

We have two major contributions in this paper. First, by encoding the data chunks according to their reference counters, we reduce the amount of data that needs to be encoded when garbage collection occurs, thus reducing the encoding overhead when data are deleted or updated in the storage system. Second, since large reference counter values usually represent high reliability or availability levels, by encoding the high/low reference counter data with respective fault tolerance erasure codes, we can provide the system with a demand reliability level and optimize the storage efficiency.

The rest of the paper is organized as follows. Section II first

978-1-5386-8367-5/18/$31.00

presents the basics of deduplication and erasure coding, two techniques which are now widely used in distributed storage systems; then it explains the performance and reliability issues of the naive erasure-coded deduplication systems with a motivational example. Section III describes our idea and algorithm. Section IV talks about the evaluation details and shows the experiment results. Section V gives the reliability analysis of RAD placement. Section VI presents related work. Section VII concludes the paper.

II. Background and Motivational Analysis

In this section, we first present the background details of the two techniques we consider in this paper, namely deduplication and erasure coding; then we introduce how recent works integrate these two techniques; and finally, we show the motivation.

A. Deduplication

As a space-saving technique, data deduplication has received broad attention recently in both academic [15]–[22] and industry [23]–[25]. Chunk-based deduplication consists of the following steps: 1) when a data object comes into the system, it will be divided into several small data chunks and each chunk will get a hash (fingerprint) by using a particular calculating algorithm, such as MD5 or SHA-1; 2) for each chunk, the system will determine whether it is a duplicate by looking it up in the hash index; 3) the new unique chunks will be stored in the system; and 4) the fingerprints of the new unique chunks will be added to the hash table and the redundant data chunks will be replaced with references to the existing chunks.

For a typical in-line deduplication system, generally we have three components on disks. 1) There is a fingerprint index, which maps the fingerprints of each stored chunk to its physical location. The fingerprint index is used to identify whether each incoming chunk is unique. 2) In addition, there is a recipe store, which maintains the logical fingerprint sequences of the data objects. Thus, when a failure occurs, we can reconstruct the data in the correct order. 3) Finally, there is a container store, which is a log-structured storage system.

After the duplicate chunks are eliminated, the unique chunks will be sealed in fixed-sized containers, which serve as disk read/write units.

B. Erasure Coding

Erasure codes provide data reliability by adding data redundancy. By combining with various coding schemes, erasure codes can achieve high data reliability with low storage overhead. In this paper, we mainly focus on the *maximum distance separable* (MDS) codes. These are configured by two parameters, n and k (where $k < n$). An (n, k) MDS code encodes k original data blocks to create $n - k$ parity blocks, so any k out of the n data and parity blocks can reconstruct all k original data blocks. The collection of the n data blocks and parity blocks is called a stripe. Usually, a stripe is distributed across n data nodes so that we can tolerant any $n - k$ node failures. Typical MDS erasure codes include Reed-Solomon codes [26], Cauchy Reed-Solomon codes [27], Local Reconstruction codes [28], and Regenerating codes [29]. In this work, we use RS-based (6, 2) codes for examples if not mentioned otherwise.

If some data blocks (no more than $n - k$) are unavailable due to node failures, reads to these unavailable data blocks are called *degraded read* because they need to retrieve data/parity blocks from other available nodes for decoding. The high performance cost of erasure coding during degraded read and update is its drawback.

C. Integration of Deduplication and Erasure Coding

As we have discussed above, deduplication systems gain storage efficiency at the cost of system reliability. After eliminating the duplicate chunks, the error resilience of the system could potentially decrease and the failure rate may increase, and this is not acceptable in real systems. In Bhagwat's work [14], this issue was first mentioned. To solve the problem, they proposed a replication-based approach, in which more popular data objects will be replicated on more devices to achieve higher reliability. Though this approach is easy to implement, it is not suitable for the recent large-scale distributed storage systems because it requires a large amount of space.

As an alternative method of replication to provide reliability, erasure coding has recently received more and more attention in the deduplication system in both academic [10], [12], [13] and industry [11]. Although the recovery process is relatively slow compared to replication, employing erasure coding can significantly reduce the storage consumption and achieve reliability.

Figure 1 displays a typical data flow in a deduplication system with erasure coding employed. Each file in the incoming file stream will go through the following steps: 1) The data stream is divided into data chunks (e.g., Rabin fingerprinting [30]). 2) The hash engine will calculate the SHA-1/MD5 digest for each chunk as its unique identification, which is called its fingerprint. 3) For each chunk, look up the fingerprint in the fingerprint index. 4) If we find a match in the fingerprint index, then the chunk is a duplicate. The chunk will be eliminated and its fingerprint will be added to the recipe store. 5) If no match is found, then the chunk is unique. The unique chunks will be erasure coded and then distributed across the storage nodes. 6) When a node writes a chunk, it first appends the chunk to an in-memory container, which will be flushed to disk when full.

D. Motivation

This is a straightforward approach to integrating erasure coding into deduplication storage systems and is used in most of the previously mentioned works. Though simple to implement, there are two potential issues in this naive approach. First, there is the issue of reliability. Erasure coding can provide the system with a certain level of fault-tolerance, but different files sometimes require different reliability levels. We can generally measure the reliability level of a file in two ways. One way is that, after deduplication, if a data chunk is

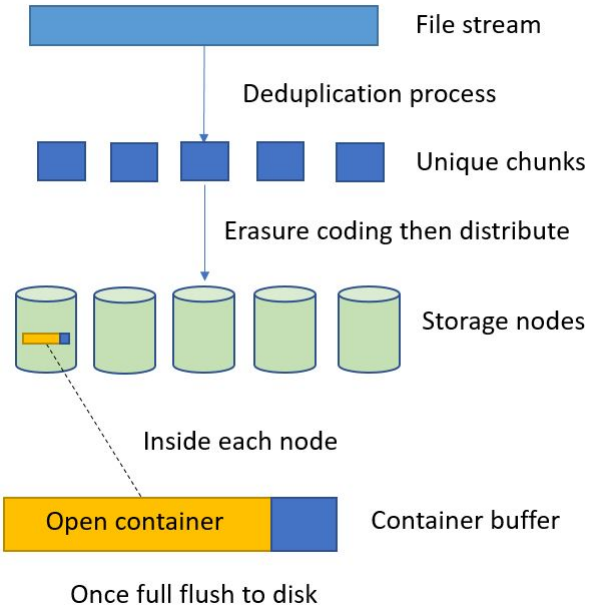
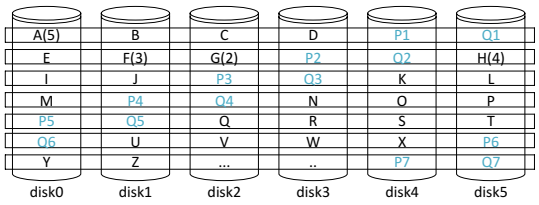


Fig. 1: Deduplication in an erasure-coded distributed storage system.

File 1	A	B	C	D	E	F
File 2	A	F	G	H	I	J
File 3	K	A	L	M	N	O
File 4	P	G	Q	R	H	S
File 5	T	U	V	W	X	A
File 6	A	F	Y	Z	H	H
.....	...	...	...	...	...	...

(a) An example of incoming data streams.



(b) An example of encoded chunks.

Fig. 2: An example of baseline erasure coding in a deduplication system.

referred to multiple times (this can be represented by a reference counter, which shows how many times this data chunk is referenced by the data files), then this chunk should be provided with a higher fault-tolerance level than data chunks with a lower reference counter (normally only one), because if this data chunk fails, multiple files will become inaccessible. The other way to measure the reliability is to check whether this data chunk has been given a certain reliability level (by the user). The second issue is the high encoding overhead when erasure coding is employed in a deduplication system. In a deduplication system, once a file is deleted or updated, the corresponding data chunks that refer to this file may become invalid (not referenced by any backup) and must be reclaimed when garbage collection occurs [10]. Recent works [31], [32] have noticed that in a deduplication system, the deletion operation is already significantly more complicated than the traditional system because of the data dependencies. However, after deploying erasure coding into the deduplication system, yet another issue arises when deletion occurs. The problem is that, after erasure coding, though one chunk is not referenced by other files, we cannot simply reclaim it because this chunk is also in one encoded data stripe which provides reliability for other data chunks. Deleting this chunk may invalidate the fault-tolerance of this data stripe. To ensure the reliability, after deleting this data chunk, the remaining chunks in the data stripe need to be encoded again. We use an example in Figure 2 to illustrate this issue.

Figure 2(a) shows an example of one incoming data stream that includes 6 data files, each of which is divided into 6 data chunks. The letter on each data chunk represents the content of that chunk. Thus, for example, the 2 A data chunks in File 1 and File 2 are identical. After chunking, the hash engine will calculate the hash number (fingerprint) of each chunk and check whether the chunk is a duplicate. If not, the chunk will be erasure coded and then written to the disks, as shown in Figure 2(b). The number in the brackets shows how many times the chunk is referenced by files (the value of the chunk's reference counter). Each column shows one coding stripe. In this example, we use RS-(6, 2) erasure codes, which means that if any 2 out of these 6 chunks fail, we can always reconstruct them by decoding. For this baseline example, we place the chunks into the storage nodes in a simple Round-Robin manner, as shown in Figure 2(a). If File 1 is going to be deleted, then chunks B, C, D, and E will be marked as invalid because they are only referenced by File 1. Chunks A and F will still remain valid because they are still referenced by other files. In this case, after chunks B, C, and D get reclaimed, only one data chunk (chunk A) and two parity chunks ($P1$ and $Q1$) will exist in the first data stripe. The stripe will no longer provide fault tolerance for the data chunk because 3 out of 6 chunks of this stripe are unavailable. To ensure the reliability, we will have to do the encoding again for data chunk A. This simple example implies that in an erasure coded deduplication system, the delete/update operation may trigger a large amount of extra encoding overhead. Further, this motivates us to exploit the connection between performing erasure coding and deduplication to more wisely encode these two layers, thus reducing the encoding overhead and providing the data with the required reliability.

III. REFERENCE-COUNTER AWARE ERASURE CODING

A. Main Idea

From the example in Section II, we find that the traditional Round-Robin placement of the chunks in a deduplication system can cause lots of extra encoding overhead. In addition, by randomly placing and encoding the chunks, the reliability of the system may not be well ensured. In this paper, we find that these two problems can both be solved by encoding the chunks according to their reference counters. In deduplication systems, each data chunk will have its own reference counter that indicates how many times this chunk is referenced by other files; once a file gets deleted, all the chunks it includes will decrement their own reference counters by 1. If we encode the chunks with high reference counters together, then it is very likely that, after reclamation, the coding stripe which includes these high reference counter chunks will still have all its chunks marked as valid. As long as the stripe remains intact, we do not need to do the extra encoding work to maintain the fault tolerance. In addition, a chunk's reference counter also represents its importance to some extent. Intuitively, we can encode the data chunks according to their reference counters to provide different levels of reliability by employing different erasure codes (e.g., we can encode the important data with high fault-tolerant erasure codes by configuring the parameters). By doing this, we can exploit the trade-off between storage efficiency and reliability among different erasure codes after they are employed in the deduplication system.

Figure 3 shows the main idea of the reference-counter aware erasure coding. 1). Files go through the chunking and deduplication process, which is the same as with the traditional deduplication system. After deduplication, chunks which have already been stored in the container store (in storage nodes) will be eliminated. 2). The unique chunks will then go to the writing buffer and get sorted by their reference counters, which will determine how they get erasure coded. Chunks with high/low reference counters will be encoded together respectively. 3). Chunks are erasure-coded into stripes. 4). After encoding, chunks will go into the container buffer and be flushed into the storage nodes when the container is full.

B. Example

In this section, we have an example demonstrating how this approach would impact the encoding operation of the system. Again, we use the figures in Section II as an example. Assume we still have 6 incoming data streams coming as shown in Figure 2(a), but this time, we encode these data chunks according to their reference-counters. Thus, chunks with higher and lower reference-counters are encoded together, respectively. Figure 4 shows the encoded chunks.

In this figure, we can see that all the chunks with reference counters higher than 1 (namely chunks $A(5), F(3), G(2)$, and $H(4)$) are encoded in the first stripe; all the other chunks with lower reference counters (in this example, lower reference-counters are all one) are encoded together in the following

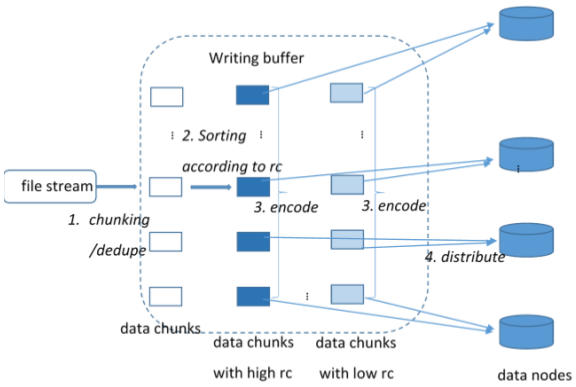


Fig. 3: Data flow of RAD.

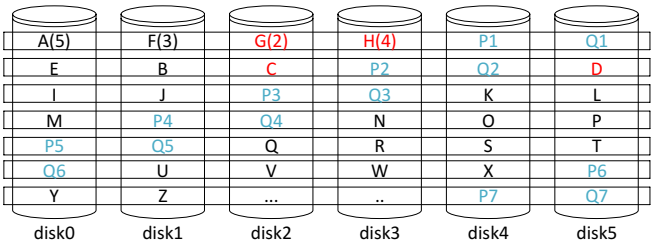


Fig. 4: Encoding according to the reference-counter.

stripes. Assume File 1 is going to be deleted. As discussed before, chunk B, C, D, and E will be marked as invalid. Chunks A and F, however, will still remain valid, as they will still be referenced by other files. However, in this case, after chunks B, C, D, and E get reclaimed, there will be no extra encoding work because stripe 1 will still have all the data chunks and stripe 2 will be deleted directly.

C. Algorithm Detail

We consider a storage system with N storage nodes and equipped with (n, k) erasure codes. Algorithm 1 shows the pseudocode of the reference-counter aware placement algorithm. For each incoming file i, it will first go through the chunking process (line 2); the number of its chunks is cn_i. All these chunks consist of u_i unique chunks and d_i duplicate chunks, so we have $u_i + d_i = cn_i$. For each chunk c_j, we calculate its fingerprint (line 5) and get its reference counter r_j. One item to note is that the duplicate chunks here differ from the traditional concept of duplicate chunks in a deduplication system. Originally, when a duplicate chunk enters the deduplication system, the chunk will be deleted after hashing and only the reference will be kept in the system. In our system, we assume the chunks are written on per-batch basis, which means we will write a certain number of chunks into the system every time. Also, the duplicate chunk we talk about in this algorithm is the chunk which is duplicate in this

specific writing-batch. Thus, the next step is to check whether the current chunk is a duplicate. If it is, then we need to know whether it has already been stored in the container store by checking the fingerprint index. If it is already stored in the storage system, then this chunk will be eliminated as the traditional deduplication system will do. If not, that means this chunk is only duplicate in this specific batch, so we increase its reference counter by 1 and later encode it with chunks which have high reference counters. If the chunk is unique, we will simply set the reference counter as 1. After this chunking and placement process, we will encode these chunks with (n, k) erasure codes and distribute them. Finally, we need to update all of the chunks' metadata to maintain the integrity of the system.

From the example we gave above, we can see that the key to decreasing the encoding overhead is to try to avoid encoding duplicate chunks and unique chunks together. In this bad case, once a file gets deleted, the unique chunks will be marked as invalid and all the duplicate chunks in the same stripe will have to be encoded again. In our scheme, the chunks in the batch will first go to the write buffer, where all the chunks will be sorted according to the value of their reference-counters. We then encode them in descending order, after which we distribute them to the storage nodes.

Algorithm 1 Reference-counter aware placement

Input: file f_i, n, k;
Output: encoded stripes;

for each file f_i **do**
  chunking f_i;
  chunks go to write-buffer (per-batch basis);
  for each chunk c_j in file f_i **do**
    calculate fingerprint for c_j;
    if c_j is duplicate **then**
      if c_j is already in container store **then**
        update the recipe store;
        r_j = r_j + 1;
        remove c_j;
      else {c_j has not been stored in container store yet}
        r_j = r_j + 1;
        c_j goes to write buffer then encode with high reference-counter chunks;
      end if
    else {c_j is unique}
      r_j = 1;
      c_j goes to write buffer then encode with low reference-counter chunks;
    end if
  end for
end for
sort the chunks according to the rc in descending order;
encode the chunks with (n, k) RS codes;
distribute the chunks across the nodes;
update the recipe and fingerprint index.

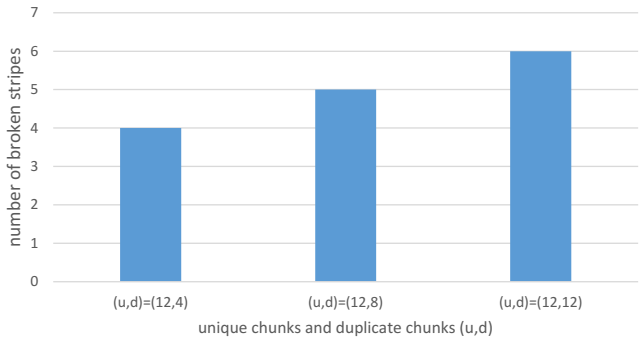


Fig. 5: Number of s_b in certain scenarios.

D. Optimization Analysis

In this section, we discuss how much better this reference-counter aware placement is than the traditional Round-Robin placement in theory. Let's first consider a scenario where one file, f_i, will be deleted. Then extra encoding will occur if any of the unique chunks in f_i is encoded with duplicate chunks in f_i or with any chunks from other files. In a typical archive operation, the number of chunks in a file is usually much larger than the number of storage nodes. Thus, we assume $u_i > k$ and $d_i > k$. Then in Round-Robin placement, all the chunks will be encoded into stripes sequentially. The number of stripes s_n will be $(u_i + d_i)/k$ (to make it simple for discussion here, we assume there is no remainder). Then the favorable case, where none of the stripes break when the deletion operation occurs, only occurs when all the unique chunks are encoded together and all the duplicate chunks are encoded together. The possibility of this is:

$$p = \frac{\prod_{n=1}^{\frac{u_i}{k}} \mathrm{C}_{nk}^{k} \times \prod_{n=1}^{\frac{d_i}{k}} \mathrm{C}_{nk}^{k}}{\prod_{n=1}^{\frac{c_i}{k}} \mathrm{C}_{nk}^{k}} \tag{1}$$

Then the *Expected Value* of the broken stripe will be $s_b = s_n \times (1 - p)$. In RAD, since we have arranged the placement of the chunks, there will ideally only be two broken stripes: one stripe which is the critical point of the unique chunk and the duplicate chunk, and the last stripe which may be encoded with chunks from other files.

The number of s_b is difficult to describe in a simple formula; thus, we have a graph in Figure 5 which shows the number of s_b in certain scenarios when N is fixed and equal to 4. From this graph, we can find that the *Expected Value* of the broken stripe is very close to the number of whole stripes, s_n, and this number increases as the chunk number, c_n, increases. Normally the chunk number of one file might reach into the hundreds or thousands, which means there exists a high possibility that we can decrease the encoding overhead by employing the RAD placement.

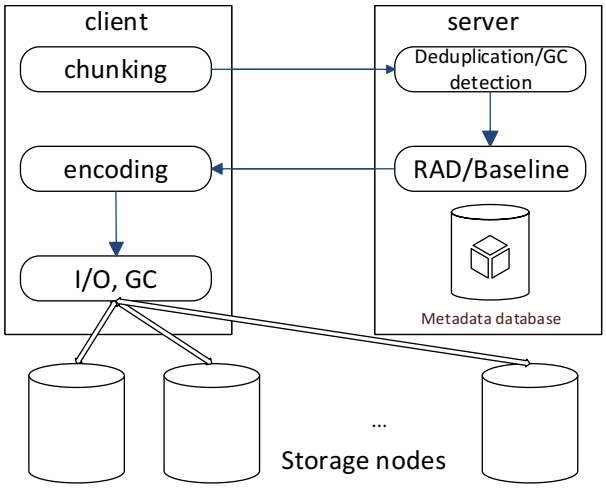


Fig. 6: The simulator architecture of RAD.

IV. Evaluation

A. System Simulator

Our evaluation is conducted by simulation and the environment is developed based on the distributed storage system simulator from [10] which realizes both deduplication and erasure coding. In our simulator, we implemented the RAD algorithm and added the garbage collection module.

Figure 6 shows the system simulator architecture which includes three main parts: the client, the metadata server, and storage nodes. In this paper, we mainly focus on the garbage collection performance when a deletion operation occurs.

For file uploads, the files are written on a per-batch basis. Each time an upload operation is initiated, the client will divide the files into chunks and get the fingerprints of these chunks. Then these fingerprints (metadata) will be sent to the metadata server, which will conduct the deduplication, run data placement algorithms (RAD/baseline), and maintain the chunk metadata in the database. Then the server responds to the client with the information of the unique chunks and the placement method. According to this information, the client will encode the unique chunks and place the chunks in the storage nodes. For garbage collection, we applied the most widely used reference count-based GC approach [33]–[35]. As we have introduced before, the reference counter of a particular chunk in a deduplication system refers to the number of times that chunk is used/referenced. A reference count value of 0 implies that the chunk is no longer shared due to users' deletion operations and can be reclaimed for garbage collection. When the client initiates a deletion operation, it will query the metadata server for all the reference counter information of the chunks in the files to be deleted. The metadata server will operate on the reference counters of these chunks and send the information back to the client, which will then reclaim the space and re-encode the chunks with reference counters greater than 1.

B. Evaluation

In this part, we will compare the GC performance of our RAD algorithm with the baseline (Round-Robin). The evaluation is conducted in the simulator introduced in Section IV.A.

1) Datasets: We divide the datasets into two categories: directories and archive/compressed files. The directories are extracted from compressed open-source project codes, such as Linux kernel codes. The directory-files usually contain multiple small files and have high similarities among different versions. The archive/compressed files are single files which usually have a large size and lower similarities between each other. The datasets we are using in our evaluation can be classified and shown in Table I. We compare each two consecutive versions of every dataset and record the similarity ratio; the average similarity is the average of all the similarity ratios. When performing garbage collection, we first upload five versions of the files, one by one; then we conduct the garbage collection of each version and compare the GC performance between the baseline scheme and RAD.

2) Simulations: The simulator builds a storage system with 16 storage nodes and deploys (n, k) = (14, 12) erasure coding. For the chunking scheme, we use 4KB fixed-size chunking.

a) Directory files: We first run the GC operation on the five directory-files; Figure 7 displays the results for the GC operation time under baseline (GCB) and RAD (GCR). From Table I, we can see that datasets LibreOffice, Linux Kernel, and Firefox directories share similar features in size (large) and similarity (high). The simulation results also show that these directories follow the same trend in their GC performance in the comparison experiments.

For LibreOffice-4.1.0.1 and 4.2.0.1, GCR shows 17.2% and 11.5% performance gains, respectively; for the other three versions of LibreOffice, however, there is not much difference. This happens because our scheme benefits more when the datasets contain more data chunks with high reference counters (greater than 1), in which case fewer stripes will be broken when GC is performed. Among the five LibreOffice datasets, the two with obvious performance gains have a larger number of high reference counter chunks than the others.

A similar conclusion can be drawn from the results of the experiments on Linux kernel and Firefox directories. When the datasets have a large number of high reference counter chunks, GCR clearly has better performance than GCB. For Linux kernel versions 2.6.31-2.6.33, GCR shows 8.6%, 22.0%, and 13.0% gains, respectively. For Firefox 20.0-50.0, GCR demonstrates 24.8%, 8.46%, 9.15%, and 13.2% gains, respectively.

For the other two datasets, Ubuntu and GDB, RAD does not feature better performance than the baseline scheme; however, this is not unexpected. Ubuntu directories are extracted from the Ubuntu ISO files, which are archive files instead of compressed files. Thus, the directories have the same level of

TABLE I: GC Dataset Description

Dataset	Description	Average Size	Average similarity	Size	Similarity
Ubuntu(D)	Ubuntu source codes directory	715.8 MB	0.1444	L(large)	L(low)
GDB(D)	GDB source codes directory	95.6 MB	0.32245	S(small)	H(high)
Firefox(D)	Firefox source codes directory	926.8 MB	0.32055	L	H
LibreOffice(D)	LibreOffice source codes directory	901.2 MB	0.359325	L	H
LinuxKernel(D)	Linux kernel source codes directory	415.2 MB	0.332275	L	H
Ubuntu(F)	Compressed Ubuntu ISO file	716.4 MB	0.1449	L	L
GDB(F)	Compressed GDB tar.gz file	17.2 MB	0.14305	S	L
Firefox(F)	Compressed Firefox tar.xz/tar.bz2 file	132.8 MB	0.1433	S	L
LibreOffice(F)	Compressed LibreOffice tar.xz file	126.6 MB	0.1475	S	L
LinuxKernel(F)	Compressed LinuxKernel tar.gz file	78.2 MB	0.1429	S	L

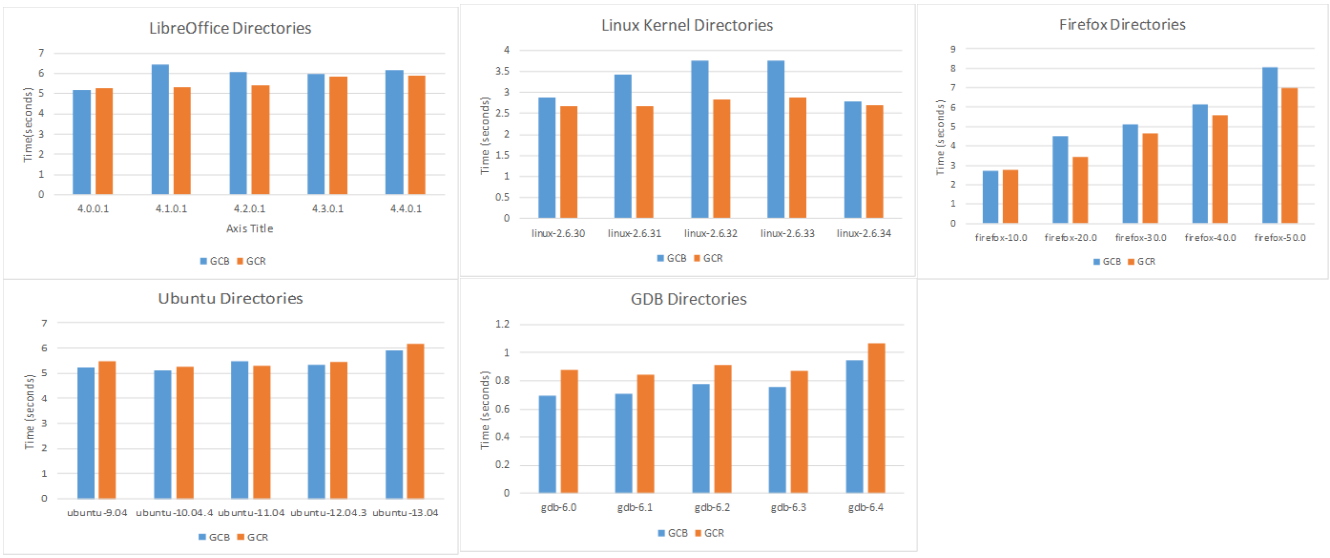


Fig. 7: GC operation time for the directory files.

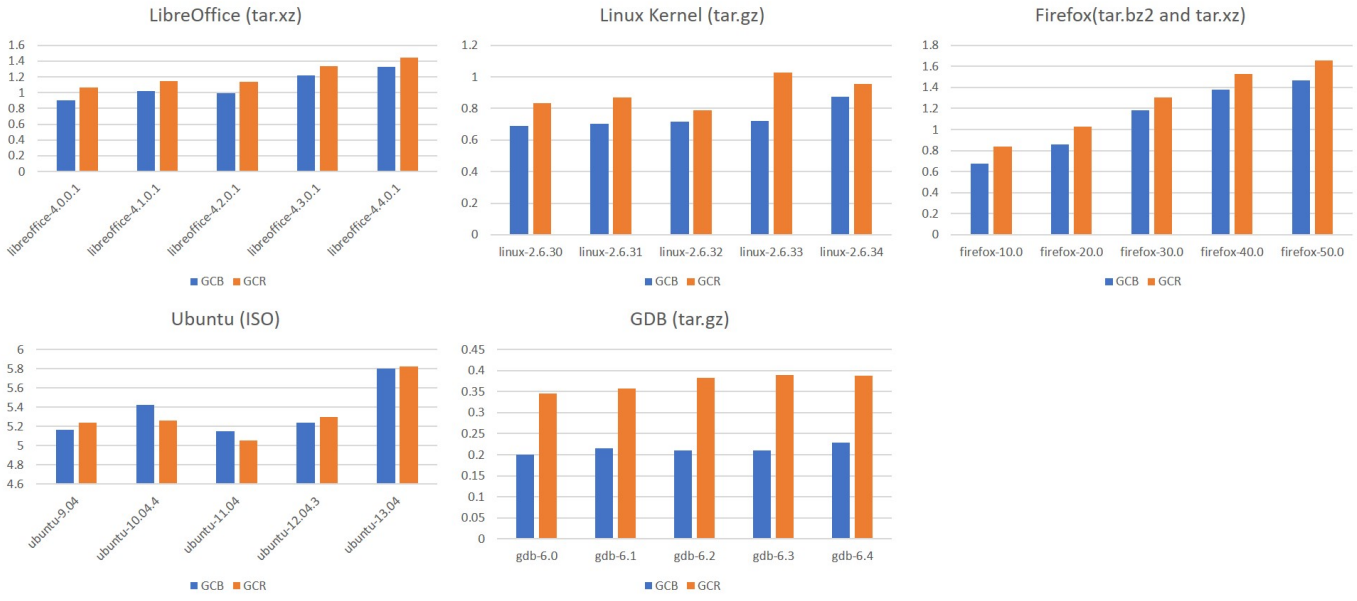


Fig. 8: GC operation time for the archive/compressed files.

similarity among the versions (around 16%), and our scheme works better for files with high similarities. For GDB, the directories' sizes are much smaller compared to the other four datasets, which means the performance degradation caused by the sorting process may offset the benefit coming from the GC performance.

b) Archive/compressed files: We then conduct the experiments on the five corresponding archive/compressed files, and results are shown in Figure 8. According to our analysis in the last section, these results match our expectations. For LibreOffice, Linux kernel, and Firefox compressed files, RAD runs a little bit longer than the baseline because these three datasets all share the same data features: small data sizes and low similarity levels. For Ubuntu ISO files, the results are similar to the results of Ubuntu extracted directory files because, as ISO files are simply archive files, the ISO files have the same data features as the extracted files. For GDB tar.gz files, the baseline clearly outperforms the RAD because the GDB file size is significantly small (around 17 MB). As a result, the GC cannot benefit much from rearranging the data chunks, and RAD's sorting process will make the running time much longer than the baseline.

V. Reliability Analysis

Several recent works [14], [36] noticed that, in a deduplication system, chunks with higher popularities deserve higher reliability, so they should be copied to maintain more replicas. In our work, chunks with high/low reference counters are encoded together respectively; thus, intuitively, there is a chance to encode these chunks with different erasure codes to provide corresponding levels of reliability and storage utilization.

Traditionally, Markov models have been used to evaluate the reliability of erasure-coded storage systems. Figure 9 shows a canonical RAID6 Markov model [37]. State 0 is the start state with all N nodes running properly. States 1 and 2 represent the system with one and two node failures, respectively. State DL ensues when a third node failure occurs, i.e., the data loss state.

The canonical Markov model can be solved analytically for MTTDL (mean time to data loss). For a storage system with N nodes and deploy erasure codes (N, K), according to [38], we have

$$MTTDL = \frac{\mu^{N-K}(K-1)}{N!\lambda^{N-K+1}} \tag{2}$$

where λ is the disk failure rate (equal to 1/MTTF) and μ is the repair rate (equal to 1/MTTR).

Assume for the baseline approach that the system encodes all the chunks with erasure codes $(B, 2)$, and in RAD we use codes $(A, 2)$ and $(C, 2)$ to encode the chunks with low/high reference counters, respectively. The storage utilization of baseline and RAD are noted by S_b and S_r, respectively, and the MTTDL are noted by M_b and M_r (expected value), respectively. Inside all the chunks, the high reference counter

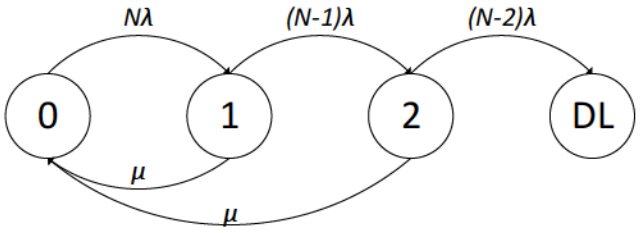


Fig. 9: Canonical RAID6 Markov model [37].

chunks account for α of the total number of chunks. Then for the storage utilization of baseline and RAD, we have

$$S_b = \frac{B-2}{B} \tag{3}$$

and

$$S_r = (1-\alpha)\frac{A-2}{A} + \alpha\frac{B-2}{B} \tag{4}$$

For the MTTDL of baseline and RAD, according to equation (2), we have

$$M_b = \frac{\mu^2}{B(B-1)(B-2)\lambda^3} \tag{5}$$

$$M_r = (1-\alpha)\frac{\mu^2}{A(A-1)(A-2)\lambda^3} + \alpha\frac{\mu^2}{C(C-1)(C-2)\lambda^3} \tag{6}$$

Assume we have MTTR as 17.8 hours and MTTF as 500,000 hours [37]. We choose four sets of erasure codes and show, in Figure 10, the storage utilization and MMTDL of the system with α ranging in $(0, 1)$. In the figure, the solid blue and red lines represent the storage utilization of baseline and RAD, respectively, and the dotted blue and red lines represent the MMTDL of the baseline and RAD, respectively. Use $(A, B, C) = (14, 10, 6)$ as an example for the analysis. For the baseline, the storage utilization and MTTDL are constants and equal to 80% and 5.48×10^{11} hours, respectively. For RAD, these two attributes change with the high reference-counter chunks ratio. From the first graph in Figure 10, we find that RAD has better storage utilization than the baseline when α is less than around 30% and has a higher MTTDL when α is greater than around 13%. So for the tuple $(A, B, C) = (14, 10, 6)$, RAD will behave better in both storage utilization and reliability when α is in the interval $(0.13, 0.30)$, which we call the R-interval. When we change the numbers of the tuple, as shown in the other 3 graphs of Figure 10, the R-interval will change but mostly still reside in $(0.05, 0.65)$. If α of a file is already known, then it will be easy to find a tuple (A, B, C) with which RAD

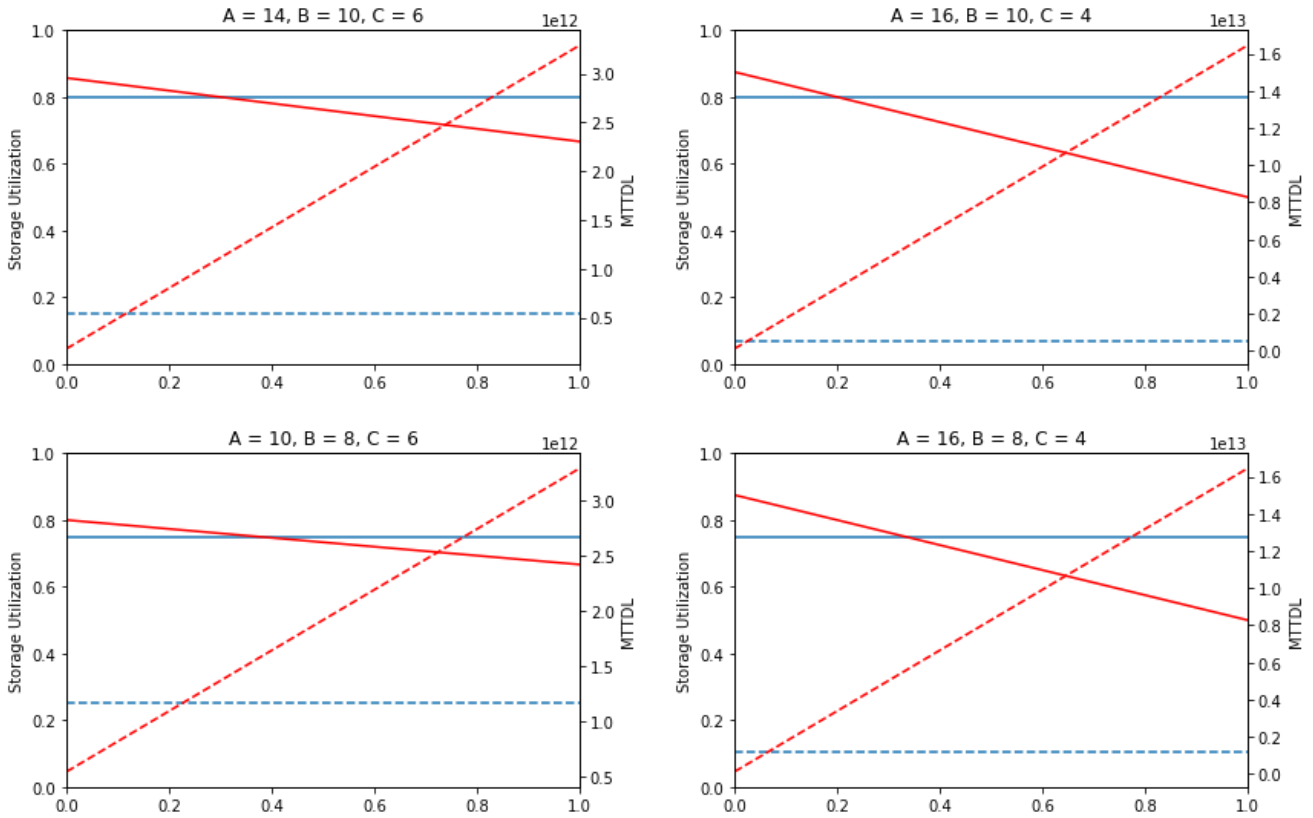


Fig. 10: Tradeoff of storage utilization and reliability under different erasure codes.

can outperform the baseline in both storage utilization and reliability. As we discussed in Section IV, Table I shows that for the normal directories and archive/compressed files, α is between $(0.14, 0.36)$, which quite suits the R-interval we get from the analysis.

VI. Related Work

As a space-saving technique, deduplication improves the space utilization; however, as a trade-off, the system reliability decreases due to the elimination of the duplicate chunks. Works have been done to ensure the reliability in deduplication system in two directions: replicate the important chunks and deploy erasure coding.

[14], [36] found that chunks with higher popularities are more important, and thus the chunks with higher reference counters should be replicated more to ensure a higher level of reliability. Some works deploy both deduplication and erasure coding in the storage system and try to solve the subsequent challenges. CodePlugin [13] introduces pre-processing steps before the normal encoding to improve the encoding performance for the redundant blocks. [39] makes the attempts to address the problem of achieving efficient and reliable key management in an erasure-coded deduplication system. [40] studies the reliability analysis of a distributed deduplication system. R-ADMAD [12] deploys deduplication with variable-size chunking over erasure-coding systems. In industry, Hydrastor [1] has deployed deduplication and erasure coding for commercial backup storage.

VII. Conclusion and Future Work

In this work, we target the garbage collection performance in an erasure-coded deduplication system. We point out that by rearranging the chunks in the deduplication storage system according to their reference counters, for most of the large-size, high-similarity files, the garbage collection performance would be increased by up to 24.8%. We also notice that after rearranging the data chunks, chunks of high-importance and low-importance are encoded together so that the system can have a customized reliability level as the users demand. Our future work will focus on deploying different erasure codes to the data chunks based on their importance level to give the system more choice in the trade-off between reliability and storage space.

The work performed is partially sponsored by the U.S. National Science Foundation grants CCF-1717660, CCF-1813081, CNS-1702474, the National Science Foundation of China (NSFC) No. 61628208, and the Shanghai National Science Foundation No. 18ZR1418500. Any opinions, findings, and conclusions or recommendations expressed in this material

are those of the authors and do not necessarily reflect the views of the funding agencies.

REFERENCES

[1] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage." in *FAST*, vol. 9, 2009, pp. 197–210.

[2] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality." in *Fast*, vol. 9, 2009, pp. 111–123.

[3] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Fast*, vol. 8, 2008, pp. 1–14.

[4] B. Andrews, "Straight talk about disk backup with deduplication," 2013.

[5] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*. IEEE, 2010, pp. 1–10.

[6] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[7] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci *et al.*, "Windows azure storage: a highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM, 2011, pp. 143–157.

[8] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "Xoring elephants: Novel erasure codes for big data," in *Proceedings of the VLDB Endowment*, vol. 6, no. 5. VLDB Endowment, 2013, pp. 325–336.

[9] H. Weatherspoon and J. D. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *International Workshop on Peer-to-Peer Systems*. Springer, 2002, pp. 328–337.

[10] M. Xu, Y. Zhu, P. P. Lee, and Y. Xu, "Even data placement for load balance in reliable distributed deduplication storage systems," in *Quality of Service (IWQoS), 2015 IEEE 23rd International Symposium on*. IEEE, 2015, pp. 349–358.

[11] X. Li, M. Lillibridge, and M. Uysal, "Reliability analysis of deduplicated and erasure-coded storage," *ACM SIGMETRICS Performance Evaluation Review*, vol. 38, no. 3, pp. 4–9, 2011.

[12] C. Liu, Y. Gu, L. Sun, B. Yan, and D. Wang, "R-admad: High reliability provision for large-scale de-duplication archival storage systems," in *Proceedings of the 23rd international conference on Supercomputing*. ACM, 2009, pp. 370–379.

[13] M. Xiao, M. A. Hassan, W. Xiao, Q. Wei, and S. Chen, "Codeplugin: Plugging deduplication into erasure coding for cloud storage." in *HotCloud*, 2015.

[14] D. Bhagwat, K. Pollack, D. D. Long, T. Schwarz, E. L. Miller, and J.-F. Pâris, "Providing high reliability in a minimum redundancy archival storage system," in *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2006. MASCOTS 2006. 14th IEEE International Symposium on*. IEEE, 2006, pp. 413–421.

[15] F. Xie, M. Condict, and S. Shete, "Estimating duplication by content-based sampling." in *USENIX Annual Technical Conference*, 2013, pp. 181–186.

[16] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design." in *USENIX Annual Technical Conference*, vol. 2012, 2012, pp. 285–296.

[17] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication." in *FAST*, 2013, pp. 183–198.

[18] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "idedup: latency-aware, inline data deduplication for primary storage." in *FAST*, vol. 12, 2012, pp. 1–14.

[19] N. Xia, C. Tian, Y. Luo, H. Liu, and X. Wang, "Uksm: swift memory deduplication via hierarchical and adaptive memory region distilling," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*. USENIX Association, 2018, pp. 325–339.

[20] F. Douglis, A. Duggal, P. Shilane, T. Wong, S. Yan, and F. C. Botelho, "The logic of physical garbage collection in deduplicating storage." in *FAST*, 2017, pp. 29–44.

[21] D. Harnik, E. Khaitzin, and D. Sotnikov, "Estimating unseen deduplication-from theory to practice." in *FAST*, 2016, pp. 277–290.

[22] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "Cachededup: In-line deduplication for flash caching." in *FAST*, 2016, pp. 301–314.

[23] "Data domain." http://www.emc.com/pdf/products/centera/centeraguide.pdf.

[24] "Quantum dxi-series." http://www.quantum.com/Products/Disk-BasedBackup/index.aspx.

[25] "Symantec puredisk." http://www.symantec.com/business/products/overview.jsp?pcid=2244&pvid=1381_1.

[26] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[27] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman, "An xor-based erasure-resilient coding scheme," 1995.

[28] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, S. Yekhanin *et al.*, "Erasure coding in windows azure storage." in *Usenix annual technical conference*. Boston, MA, 2012, pp. 15–26.

[29] K. Rashmi, N. B. Shah, K. Ramchandran, and P. V. Kumar, "Regenerating codes for errors and erasures in distributed storage," in *Information Theory Proceedings (ISIT), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 1202–1206.

[30] M. O. Rabin *et al.*, *Fingerprinting by random polynomials*. Center for Research in Computing Techn., Aiken Computation Laboratory, Univ., 1981.

[31] P. Strzelczak, E. Adamczyk, U. Herman-Izycka, J. Sakowicz, L. Slusarczyk, J. Wrona, and C. Dubnicki, "Concurrent deletion in a distributed content-addressable storage system with global deduplication." in *FAST*, 2013, pp. 161–174.

[32] D. N. Simha, M. Lu, and T.-c. Chiueh, "A scalable deduplication and garbage collection engine for incremental backup," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013, p. 16.

[33] J. Wei, H. Jiang, K. Zhou, and D. Feng, "Mad2: A scalable high-throughput exact deduplication approach for network backup services," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–14.

[34] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information." in *USENIX Annual Technical Conference*, 2014, pp. 181–192.

[35] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system." in *USENIX annual technical conference*, 2011.

[36] D. Bhagwat, K. Eshghi, D. D. Long, and M. Lillibridge, "Extreme binning: Scalable, parallel deduplication for chunk-based file backup," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems, 2009. MASCOTS'09. IEEE International Symposium on*. IEEE, 2009, pp. 1–9.

[37] I. Iliadis, "Reliability of data storage systems." https://www.iaria.org/conferences2015/filesCTRQ15/IliasIliadis_CTRQ_2015_Keynote.pdf, 2015.

[38] A. Thomasian, "Mirrored and hybrid disk arrays: Organization, scheduling, reliability, and performance," *arXiv preprint arXiv:1801.08873*, 2018.

[39] J. Li, X. Chen, M. Li, J. Li, P. P. Lee, and W. Lou, "Secure deduplication with efficient and reliable convergent key management," *IEEE transactions on parallel and distributed systems*, vol. 25, no. 6, pp. 1615–1625, 2014.

[40] Y. Nam, G. Lu, and D. H. Du, "Reliability-aware deduplication storage: Assuring chunk reliability and chunk loss severity," in *Green Computing Conference and Workshops (IGCC), 2011 International*. IEEE, 2011, pp. 1–6.