

Received 5 March 2023, accepted 12 March 2023, date of publication 15 March 2023, date of current version 10 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3257404

RESEARCH ARTICLE

BPR: An Erasure Coding Batch Parallel Repair Approach in Distributed Storage Systems

YING SONG^{1,2,3}, WENXUAN ZHAO^{1,2}, AND BO WANG⁴

¹Beijing Key Laboratory of Internet Culture and Digital Dissemination, Beijing Information Science and Technology University, Beijing 100101, China

²Beijing Advanced Innovation Center for Materials Genome Engineering, Beijing Information Science and Technology University, Beijing 100101, China

³State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100086, China

⁴Software Engineering College, Zhengzhou University of Light Industry, Zhengzhou 450002, China

Corresponding author: Ying Song (songying@bistu.edu.cn)

This work was supported in part by the National Natural Science Foundation of China under Grant 61872043; and in part by the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), under Grant CARCHA202103.


ABSTRACT Today, Erasure Coding is one of the most significant techniques widely used in distributed systems because it can improve reliability for large amounts of data with low storage overhead. However, when the distributed system encounters a large number of data loss in stripes and requires batch-stripes data recovery, current data recovery methods either repeat the single-stripe recovery method or only optimize partial stripe recovery when recovering large-scale stripes, which incurs heavy upload and download repair traffics and imbalanced load, affecting the efficiency of fault recovery and wasting additional resources. In this paper, we propose BPR, an Erasure Coding batch parallel repair approach for distributed storage systems. BPR reduces cross-rack network transfer time and increases recovery throughput by classifying the stripes and recovering the data of stripes in batches through the forward and reverse parallel data recovery. The experiment results show that for large-scale stripes recovery, BPR reduces the cross-rack network transfer time by up to 10% and increases the recovery throughput by up to 8% compared with the rPDL in some scenarios.

INDEX TERMS Distributed storage system, erasure coding, data recovery.

I. INTRODUCTION

Large-scale distributed clusters, such as Hadoop, are usually composed of many independent low-reliability commercial components, and unexpected failures of components are common. To ensure high reliability and availability of data in such a distributed storage system, a common approach is to use three-way replication redundancy technology, which provides fault tolerance by storing multiple copies of the same data on different nodes. When the amount of data is small, the replication technique is simple and effective. But in large data centers, three-way replication technology requires twice the storage overhead, which is quite expensive and unacceptable.

As an alternative, Erasure Coding can provide fault tolerance which close to replication technology with lower storage

The associate editor coordinating the review of this manuscript and approving it for publication was Chong Leong Gan .

overhead. Among multiple Erasure Coding families, Reed-Solomon code [1] is the most widely used code. An $RS(n, m)$ encodes n data chunks into m parity chunks, and these $n + m$ independent chunks form a stripe. These chunks of each stripe are stored in vary racks called the *host* racks for the stripe. By decoding any n of the remaining available chunks in the stripe, failed chunks less than or equal to m can be recovered. However, using Erasure Coding to recover failed chunks in a stripe requires retrieving available chunks in multiple *source* racks, which offer chunks to recover chunks through cross-rack transfer, and incurs high recovery costs. Although Erasure Coding can improve storage efficiency, they significantly increase disk I/O and network bandwidth occupation. Erasure coding has been widely adopted in cloud storage systems, e.g., Azure [20], Facebook [21] and distributed storage systems [9], [17], [20].

In modern large-scale data centers, storage nodes are divided into different racks. Storage nodes are connected

in a same rack through the switch. And, multiple racks are then interconnected via the network core [2], [3]. In this architecture, if you want to repair data of stripes, you need to transfer data between multiple rack, and the cross-rack bandwidth is often competed among the racks. In addition, the available cross-rack bandwidth of each node is only 1/20 to 1/5 of the intra-rack bandwidth [4]. Therefore, cross-rack bandwidth is much scarcer than intra-rack bandwidth [5], and excessive cross-rack bandwidth slows down the recovery process.

To mitigate the impact of cross-rack data repair, existing works design new repair scheduling methods to minimize cross-rack repair traffic [7], [8], [12], [14]. However, these data recovery methods still face a non-trivial challenge. The challenge is that the traffic of upload and download can't be well balanced when repairing large amounts of stripes. Some works begin to consider such reality and balance upload and download traffics by means of substituting and swapping repair solution, such as ClusterSR [12]. However, when chunks are stored centrally on several racks, these works will not avoid selecting the R_f with the failed node as the *destination* rack. When it selecting the R_f with the failed node as the *destination* rack, R_f will suffer from much heavier cross-rack traffic than the others when recovering large-scale stripes. Xu et al. [14] have considered this question about R_f , but this work repeats the recovery method which recovers a single stripe so that some destination racks possibly suffer download traffic congestion when recovering large-scale stripes.

In order to scatter and balance the traffic of upload and download during the recovery of large amounts of stripes, in this paper, we propose BPR, an Erasure Coding batch parallel repair approach for the distributed storage system. BPR focuses on classifying the stripes and dividing recovery operations into some small partial symmetry operations for benefits of full-duplex transmission. First and foremost, BPR distributes all chunks of the stripes into some racks and then provides a stripe classification scheme. At last, BPR constructs multi-stripe repair solutions based on the classification results. Through forward and reverse parallel transfer, BPR can significantly reduce cross-rack data transfer time and increase recovery throughput in an Erasure Coding distributed storage system.

Our contributions are summarized as follows:

- We present BPR, an Erasure Coding batch parallel repair approach. It can recover data of stripes in batches and effectively reduce cross-rack data transmission time and total recovery time, greatly increase recovery throughput in the event of chunks failed in the Erasure Coding distributed storage system.
- In order to effectively recover data in batches, we also propose a stripe classification method, which divides strips into different Batches_{single} according to a unified classification method for data recovery.
- We conduct a group of experiments to evaluate BPR. The results show that compared with rPDL, BPR reduces

up to 10 % cross-rack network transmission time, and increases recovery throughput by up to 8%.

The rest of this paper is summarized as follows. Section II provides background and some preliminary work. Section III introduces the latest work. Section IV introduces some problems we have observed. Section V introduces the stripe classification method and BPR design. Section VI makes theoretical analysis of BPR. Section VII evaluates the performance of BPR. Section VIII summarizes this work.

II. BACKGROUND

In this section, we briefly review the basic properties of Erasure Coding and the related work of partial decoding, which are the basis of our recovery method.

A. ERASURE CODING

This paper focuses on a famous Reed-Solomon (RS) code. Compared with replication storage, Erasure Coding storage provides similar fault tolerance, but the storage cost is greatly reduced. An RS code usually has two parameters, n and m , which represent that an RS code encodes n into m parity chunks, so that n data chunks and m parity chunks form a stripe. Any remaining n chunks can be decoded to repair the original data chunk, so that at most m data chunk failures can be tolerated. Figure 1 presents the details of the matrix coding process of a RS(5, 3) code. Here, data chunks and parity chunks are organized into fixed-size units called chunks.

If we place n chunks of each stripe on n different data nodes, the system can tolerate the node-level failure. Because then, if one rack is damaged, we can find $n-m$ chunks from other racks where data is stored to participate in data recovery.

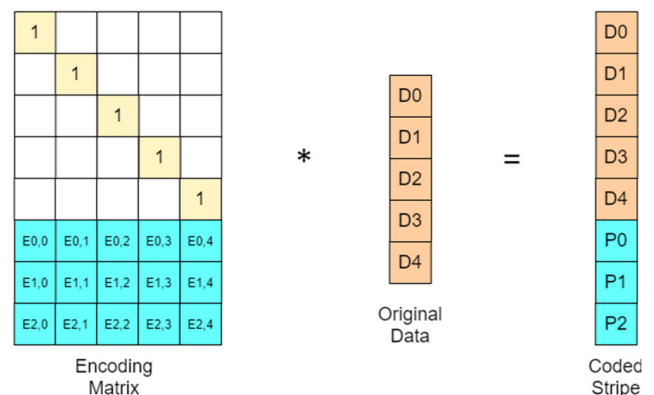


FIGURE 1. Encoding process of an RS (5, 3) code.

B. PARTIAL DECODING

In actual production, the inner-rack bandwidth is 10Gb/s, but the cross-rack bandwidth is only 1Gb/s [9], so the cross-rack bandwidth is extremely scarce. Then, the traditional data recovery transfer all the chunks involved in the recovery in the stripe to the *destination* racks which store the new repaired chunk, which generates a lot of cross-rack traffic

jams, greatly prolongs the total recovery time and causes load imbalance. To solve this problem, we can distribute the decoding and network transfer processes to other racks, and these operations can be performed in parallel over a period of time, thus reducing the data transfer time.

On the other hand, in the encoding/decoding process, the RS code adopts the four algorithms defined in Galois Field $GF(2^w)$. The $GF(2^w)$ field has 2^w values, each of which corresponds to a polynomial of degree less than w , so that the four operations on the field are converted into operations in the polynomial space. An important property of the GF algorithm is that addition is XOR. Therefore, RS codes are encoded/decoded by mathematical linear combination. Therefore, we can partially encode the chunks involved in recovery in the rack in advance to generate an intermediate coding chunk whose size is the same as the size of the data chunk. Therefore, in the process of data transmission, the number of chunks that need to be transmitted across racks will be greatly reduced, and the total traffic transmitted across racks will be reduced, so the time of network transmission across racks can be significantly reduced. Although there will be a few more decoding processes, the total repair time is greatly reduced because the cross-rack bandwidth is about 10 times the bandwidth within the rack.

III. RELATED WORK

With Hadoop3 becoming more dominant, there are more and more solutions using Erasure Coding strategies. So far, a lot of research has focused on the data recovery problem of Erasure Coding, and the results are also widely used in distributed systems such as Hadoop. According to different research motivations, we briefly review the latest scientific research achievements of Erasure Coding storage systems as follows.

To reduce the data transfer for data updates in Erasure Coding distributed systems, Hu et al. [10] proposed a data update scheme based on the Ant Colony Optimization, ACOUS. They employed a two-stage rendezvous data update procedure to optimize the multiple data nodes updates. On the other hand, for improving the storage efficiency of Erasure Coding storage systems, Zhang and Hu [11] employed novel scaling methods for E-MBR and Butterfly codes, and utilize parity locally update and features of the two codes to reduce the bandwidth cost.

More significantly, we focus on reducing the data transfer for data recovery. Shen [12] proposed ClusterSR, a cluster-aware decentralized repair method. He mainly focuses on decentralized repair scenarios, observing that due to the wide support for full-duplex transmission, data can be independently sent and received at the same transmission rate in actual production, by balancing cluster upload and download traffic, to reduce repair traffic across clusters. However, to maximize the advantage of partial decoding for saving cross-rack repair traffic, chunks are stored centrally on several racks. When single node failure occurs, in order to obtain m -node fault tolerance for $RS(n, m)$, ClusterSR will not avoid selecting the R_f with the failed node as the *destination* rack.

When it selecting the R_f with the failed node as the *destination* rack, R_f will suffer from much heavier cross-rack traffic than the others when recovering large-scale stripes.

Shi and Lu [13] propose a set of coherent in-network EC primitives, named INEC, which can be integrated into five EC schemes. And, INEC can significantly reduce latencies, and accelerate the throughput, write, and degraded read performance. Xu et al. [14] proposed a restoration method rPDL based on PDL data layout. rPDL proposes two restoration schemes R-w-PD and R-w/o-PD for PDL data layout and aiming at reducing cross-rack traffic. R-w-PD transmits chunks directly to the rack which stores new repaired chunk, while R-w/o-PD uses partial decoding to reduce the number of chunks that need to be transmitted. Then compare the cross-rack traffic required by R-w-PD and R-w/o-PD in the scenarios, and choose a recovery plan with low traffic. Finally, fault recovery is realized with high parallelism, which reduces the time cost of cross-rack recovery. However, this work randomly selects destination racks so that some *destination* racks possibly suffer serious download traffic congestion. Bai et al. [15] propose PPT and PPCT, which use a special bandwidth gap to transfer data parallel to bypass low-bandwidth links. But this method will bring network congestion and competition. Zhou et al. propose SMFRepair [16], a single-node multi-level forwarding repair technique. SMFRepair carefully selects the helper nodes and uses idle nodes to bypass low-bandwidth links. Idle nodes have sufficient and unused network bandwidth. It also pipelines the repair links that are optimized by idle nodes. However, SMFRepair rarely take into account the multi-stripes repair and total repair time is not always the least time.

IV. OBSERVATIONS AND PROBLEM

Given the scarcity of the cross-bandwidth, we have the following observations. For short, we call the i th stripe S_i and the i th rack R_i . In order to introduce our observations, we use four stripes labelled by $\{S_1, S_2, S_3, S_4\}$ and six racks labelled by $\{R_1, R_2, R_3, R_4, R_5, R_6\}$ as a storage system instance as illustrated in Figure 2(a). These racks are numbered by the position. For S_1 , racks $\{R_2, R_3, R_4, R_5\}$ store all chunks of this stripe, and we call these four racks *host* racks for S_1 . Racks $\{R_1, R_6\}$ don't store chunks of S_1 so we call these two racks *non-host* racks for S_1 . In order to tolerate single-rack failures for $RS(n, m)$, we would place at most m chunks of the same stripe to the same rack. Meanwhile, racks $\{R_2, R_3, R_4, R_5\}$ all exactly store four chunks of S_1 in the $RS(12, 4)$ so these four *host* racks is *full* racks for S_1 . Otherwise, the *host* racks which store less than four chunks of a stripe are called *non-full* racks for this stripe.

According to the data layout in Figure 2(a), we assume that each stripe of the four $RS(12,4)$ stripes has one failed chunk as illustrated in Figure 2(b), and the system needs to restore all these failed chunks. We use R_f to denote the rack which stores the failed chunk for a stripe. The rack which stores the new repaired chunk of a stripe is called *destination* rack.

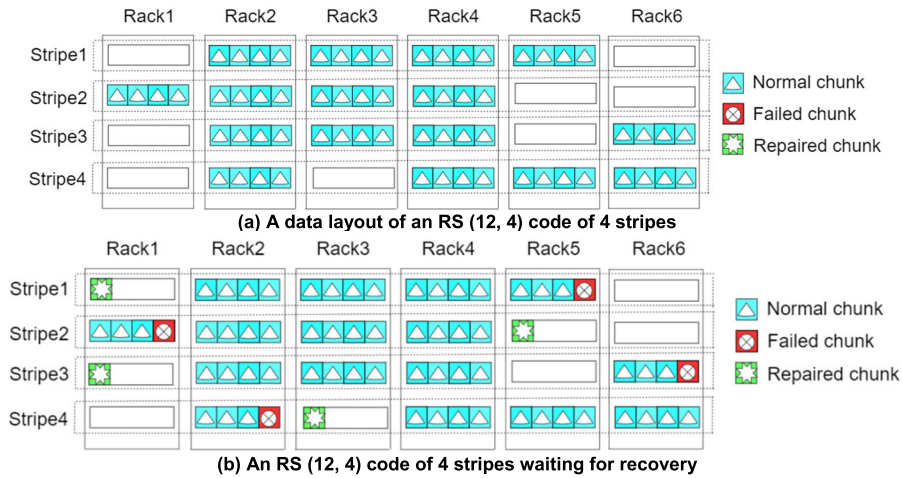


FIGURE 2. Examples of a data layout of an RS (12, 4) code of 4 stripes.

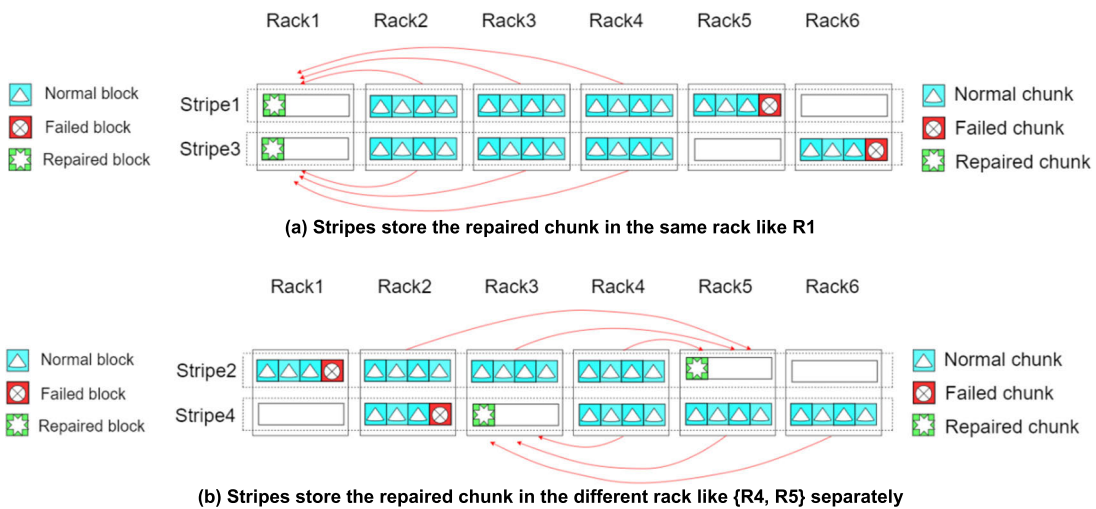


FIGURE 3. Examples of recovering four stripes.

In the existing recovery method, if all *host* racks are *full*, the system randomly selects a *non-host* rack as a *destination* rack to store new repaired chunks. For $S1$, these three *host* racks $\{R2, R3, R4\}$ offer chunks to recover chunks through cross-rack transfer, we call these racks *source* racks. Meanwhile, to avoid Rf suffers from much heavier cross-rack traffic than the other racks [16], we don't select the Rf as the *destination* rack and *source* rack.

In the regular repair strategy, there are two observations.

Observation1: As shown in Figure 3(a), the repair solution retrieves 12 surviving chunks from $\{R2, R3, R4\}$ in the $\{S1, S3\}$, and stores two new repaired chunks in the same rack $R1$. To avoid the rack Rf which stores the failed chunk suffering from much heavier cross-rack traffic than the other racks, racks $\{R5, R6\}$ will not take part in the reconstructions of stripes $\{S1, S3\}$. Through the partial decoding technology, a stripe only needs to aggregate and transmit three chunks from racks $\{R2, R3, R4\}$ to $R1$ for repair. But if the two stripes

are recovered at the same time, $R1$ needs to download six chunks at once, and its serious congestion of download traffic will greatly increase the total time of network transmission.

Observation2: As shown in Figure 3(b), the repair solution retrieves 12 surviving chunks from racks $\{R2, R3, R4\}$ and racks $\{R4, R5, R6\}$ in the stripes $\{S2, S4\}$, and stores repaired chunks in the different rack like racks $\{R3, R5\}$.

In this case, the download traffic of chunks is scattered to racks $\{R3, R5\}$, but $R4$ still has to transmit four chunks to other racks considering Observation1 and Observation2 at the same time. The upload traffic of chunks is still blocked, and the total time of the network transmission also increases.

We find that once most stripes need to be recovered at the same time, multiple stripes may select same *non-host* racks to store new repaired chunks and the same *source* racks to transmit data together at the same time, resulting in download or upload traffic congestion of the single or multiple racks. The total time of network transmission is greatly increased.

From Observation1 and Observation2 we learned that solving the problem of traffic congestion of upload and download is the key to reducing the total time of network transmission so as to greatly reduce the total repair time cost.

V. THE DESIGN OF BPR

In order to solve the above mentioned problem of upload and download traffic congestion, we propose an Erasure Coding batch parallel repair method, BPR. BPR focuses on classifying the stripes and divide recovery operations into some small partial symmetry operations for benefits of full-duplex transmission. BPR includes the process of two phases, namely the chunks initial layout phase and the failed chunks recovery phase. In the first phase, BPR distributes all chunks of stripes into various racks when the data is initial stored in the storage system. In the second phase, in order to perform efficient batch recovery, BPR classifies the large-scale stripes to each small batch called $\text{Batch}_{\text{single}}$, in which stripes can be restored at the same time, and then BPR implements different recovery schemes based on the characteristics of different $\text{Batches}_{\text{single}}$, with the goal of scattering and balancing cross-rack upload and download traffic.

A. DISTRIBUTING CHUNKS OF STRIPES

BPR is applied to the $\text{RS}(n,m)$. When placing chunks, in order to obtain m -node fault tolerance, we should place at most m chunks of the same stripe to the same rack and any two chunks of the same stripe are not distributed to the same node. Then to maximize the advantage of partial decoding, BPR places as many chunks of the same stripe to the same rack as possible. For the uniform distribution of chunks, the numbers of chunks of a same stripe in any two racks differ by at most 1. $N(n,m) = \lceil \frac{n+m}{m} \rceil$ is the number of *host* racks which store chunks of the same stripe.

$$N(n, m) = \lceil \frac{n+m}{m} \rceil = k \quad (1)$$

For example illustrated in Figure 2(a), in the process of $\text{RS}(12,4)$ placement, BPR places all 16 chunks of the same stripe on four racks and each rack hosts four of the 16 chunks.

In addition, in the actual production, the probability of the single failure for each stripe is more than 90% [6], [17], [19]. Therefore, BPR mainly focuses on the single failure repair. BPR is applicable for the single-chunk failure and the single-node failure.

B. CLASSIFYING STRIPES

In order to standardize recovery according to different stripes' distributions, we divide the stripes to be recovered into the smallest unit of batch recovery, $\text{Batch}_{\text{single}}$, and then execute recovery scheme for each $\text{Batch}_{\text{single}}$.

Algorithm 1 describes how to select *destination* racks for stripes which has *non-full host* racks and divide all the stripes into each $\text{Batch}_{\text{single}}$ based on *source* racks and the R_f . Firstly, if there is a *non-full host* rack selected as the *destination* rack

for a stripe, BPR doesn't need to transfer chunks of this *non-full host* rack through cross-rack so as to reduce cross-rack traffic. Therefore, if a stripe has *non-full host* racks, BPR selects a random *non-full host* rack as the *destination* rack to save the new repaired chunk for this stripe. Meanwhile, BPR selects other *non-full host* racks and *full host* racks as *source* racks to transfer chunks for recovery this stripe.

We observe the situation that for the full-duplex transmission, the process of that R_4 transfers chunks to R_3 in S_1 and R_3 transfers chunks to R_4 in S_2 is a symmetry and parallel cross-rack transfer process when we recovery two stripes in Figure 6(a) at the same time. That transfer process rarely causes upload and download traffic congestion. These two stripes $\{S_1, S_2\}$ have the same *source* racks and R_f . In order to implement this process for a large-scale stripe to be recovered, BPR firstly divide the stripes with the same *source* racks and R_f into $\text{Batches}_{\text{single}}$.

Algorithm 1 Classifying Stripes.

Input: Failed stripes
Output: $\text{Batch}_{\text{single}}$

```

1: Initialize a set of non-full stripes as  $\text{nfStripeSets} \{S_1, S_2 \dots S_n\}$ ;  $R_{di}$  is the destination rack for  $S_i$ 
   a set as  $\text{Batch}_{\text{odd}}$ ; a set as  $\text{Batch}_{\text{even}}$ ; a set of all stripes as  $\text{aStripeSets} \{S_1, S_2 \dots S_x\}$ ;
2: //Selecting destination racks for non-full stripes
3: For each stripe  $S_i$  in  $\text{nfStripeSets}$  do
4:    $R_{di} = \text{random}(\text{non-full source racks in } S_i)$  4: //Start to sort stripes
5: For stripes in  $\text{aStripeSets}$  do
6:   Select stripes which have same source racks and the  $R_f$  to form into  $\text{Batch}_{\text{single}}$ 
7: For each  $\text{Batch}_{\text{single}} B_i$  do
8:   if  $\text{size}(\text{source racks in } B_i) \% 2 = 1$  then
9:     Add  $B_i$  to the  $\text{Batch}_{\text{odd}}$ 
10:  else
11:    Add  $B_i$  to the  $\text{Batch}_{\text{even}}$ 

```

Next, we observe the R_5 has to transfer chunks to other racks individually in Figure 6(b) than all *source* racks in Figure 6(a) when we recovery S_1 and S_2 in Figure 6(b). The number of *source* racks is even for two stripes in Figure 6(a) and the number of *source* racks is odd in Figure 6(b). Therefore, we need to implement different recovery solutions to recovery these two types of stripes in Figure 6(a) and Figure 6(b). BPR secondly divide these $\text{Batches}_{\text{single}}$ to the $\text{Batch}_{\text{even}}$ or the $\text{Batch}_{\text{odd}}$.

Stripes are divided into the same $\text{Batch}_{\text{single}}$ respectively, depending on whether the *source* racks and R_f are the same. For example, Figure 4 shows that S_1 has three racks $\{R_3, R_4, R_5\}$ whose chunks are to be involved in cross-rack transfer, so these three racks are the *source* racks for S_1 . For the R_2 of S_5 is a *non-full host* rack which is selected as the *destination* rack to save a new repaired chunk for this stripe so that S_5 has two *source* racks $\{R_3, R_4\}$. The *source* racks of stripes $\{S_1, S_2\}$ are the same racks $\{R_3, R_4, R_5\}$ and R_f is the same rack R_2 , so $\{S_1, S_2\}$ form a $\text{Batch}_{\text{single}}$. And the number of *source* racks is three, an odd number, so this $\text{Batch}_{\text{single}}$ belongs to the $\text{Batch}_{\text{odd}}$.

C. RECOVERING STRIPES

In the above, we have classified the stripes to be repaired and divided them into $\text{Batches}_{\text{single}}$, and then we will restore

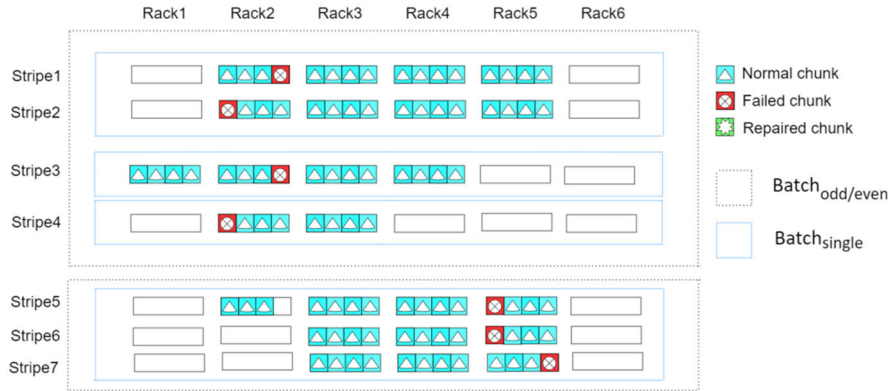


FIGURE 4. Examples of classifying stripes with BPR.

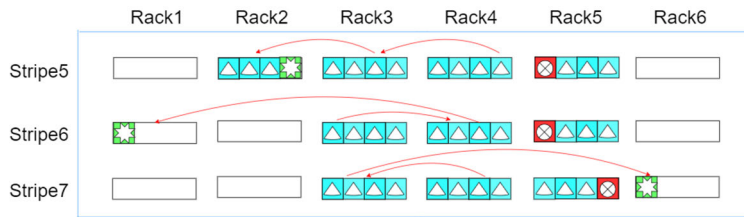


FIGURE 5. Examples of recovering stripes with BPR.

these $\text{Batch}_{\text{single}}$, Algorithm 2 elaborates the recovery detail procedures.

Algorithm 2 Recovering stripes.

Input: a $\text{Batch}_{\text{single}}$
Output: a valid transfer solution for the $\text{Batch}_{\text{single}}$

```

1: Initialize a set of destination racks of non-full stripes  $\text{RackSets}$  { IDs of racks : num;... };
   a set of full stripes as  $\text{StripeSets}$  {  $S_1, S_2, \dots, S_n$  };  $R_{di}$  is the destination rack for  $S_i$ ;
   a set of non-host racks as  $\text{NRackSets}$  {  $R_1, R_2, \dots, R_x$  };  $NsNum = \text{size}(\text{NRackSets})$ ;
   a set of source racks as  $\text{SRackSets}$  {  $R_1, R_2, \dots, R_t$  }; Function  $\text{Transfer}(R_1, R_2, R_{di})$ ;

2://Selecting destination racks for full stripes
3:  $\text{Int num} = 0$ ; // the index of the rack
4: For each stripe  $S_i$  in  $\text{StripeSets}$  do
5:   While ( $\text{NRackSets}[\text{num}] \neq \text{RackSets}[\text{num}]$  &&  $\text{RackSets}[\text{num}] > 0$ ) {
6:      $\text{RackSets}[\text{NRackSets}[\text{num}]] = 1$ ;  $\text{num} = (\text{num} + 1) \% NsNum$ ;
   }
7:  $R_{di} = \text{NRackSets}[\text{num}]$ ;  $\text{num} = (\text{num} + 1) \% NsNum$ 
8://Execute Batchevenodd recovery method
9://Transfer ( $R_1, R_t, R_{di}$ ) means: Transfer chunks from  $R_1 \dots R_t$ , then to destination rack  $R_{di}$ 
10:// Transfer ( $R_1, \text{null}, R_{di}$ ) means: Transfer chunks directly from  $R_1$  to destination rack  $R_{di}$ 
11://Transfer chunks in Batcheven
12: For each stripe  $S_i$  in  $\text{Batch}_{\text{single}}$  do
13:   if ( $i \% 2 = 1$  &&  $\text{Batch}_{\text{single}} \text{Batcheven}$ )
14:     Transfer( $R_i, R_1, R_{di}$ )
15:   else if ( $i \% 2 = 0$  &&  $\text{Batch}_{\text{single}} \text{Batcheven}$ )
16:     Transfer ( $R_i, R_t, R_{di}$ );
17://Transfer chunks in Batchodd
18:   if ( $i \% 2 = 1$  &&  $\text{Batch}_{\text{single}} \text{Batchodd}$ )
19:     Transfer ( $R_i - 1, R_1, R_{di}$ );
20:     Transfer ( $R_i, \text{null}, R_{di}$ )
21:   else if ( $i \% 2 = 0$  &&  $\text{Batch}_{\text{single}} \text{Batchodd}$ )
     Transfer ( $R_i, R_t - 1, R_{di}$ );
     Transfer( $R_t, \text{null}, R_{di}$ );

```

Algorithm 2 describes how to select *destination* rack and recover the stripes of the $\text{Batch}_{\text{single}}$. We call the stripes which have *non-full host* racks *non-full* stripes and the stripes whose *source* racks is all *full host* racks *full* stripes.

Non-full stripes have already selected *non-full host* racks as the *destination* racks and *full* stripes will select *non-host* racks as the *destination* racks. To scatter and balance the global download traffic, BPR starts to select the *destination* racks for stripes of every $\text{Batch}_{\text{single}}$ in a specific way. BPR first gathers the number and IDs of all *destination* racks selected by *non-full* stripes in a $\text{Batch}_{\text{single}}$. Then BPR numbers the *full* stripes and *non-host* racks in an ascending order of IDs. This *non-host* racks are the same for the all *full* stripes in a $\text{Batch}_{\text{single}}$. At last, the *full* stripes select racks from *non-host* racks numbered as the *destination* racks in an ascending order. If there are *non-full* stripes which have already selected this rack once, BPR skips selecting this rack once and selects next rack orderly to save this new repaired chunk.

For example, as shown in Figure 4 and Figure 5, stripes $\{S_5, S_6, S_7\}$ form into a $\text{Batch}_{\text{single}}$. S_5 is a *non-full* stripe and $\{S_6, S_7\}$ are two *full* stripes. S_5 has already selected a *non-full* rack as a *destination* rack. BPR gathers the information of the *destination* rack for this *non-full* stripe, i.e., {ID: number} like $\{R_2:1\}$, which means that R_2 has been selected as the *destination* rack and suffered download traffic once. To reduce the download traffic of R_2 , we will reduce one chance to act as the *destination* rack in this $\text{Batch}_{\text{single}}$. Meanwhile, stripes $\{S_6, S_7\}$ need to select *non-host* racks as *destination* racks. BPR numbers the S_6 and S_7 from 1st to 2nd and numbers the $\{R_1, R_2, R_6\}$ from 1st to 3rd. Then, we start to select *non-host* racks $\{R_1, R_2, R_6\}$ to $\{S_6, S_7\}$. In an ascending order, we first select 1st of *non-host* racks to 1st of *full* stripes and select 2nd of *non-host* racks to 2nd

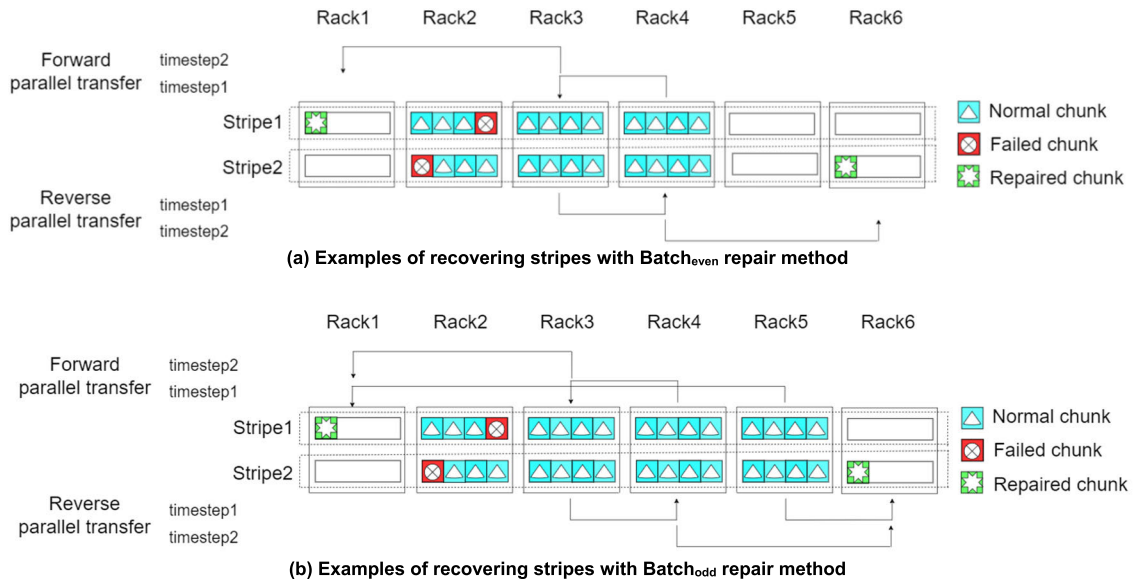


FIGURE 6. Examples of recovering stripes with $Batch_{even}$ repair method and $Batch_{odd}$ repair method.

of full stripes, which means we first select $R1$ to $S6$ and $R2$ to $S7$. However, in terms of $\{R2:1\}$, the non-full stripe $S5$ has already select $R2$ as the destination rack once. We skip selecting $R2$ and select the next non-host rack $R6$ to $S7$.

Then, in order to scatter the upload and download traffics from the source racks and destination racks during data transfer, we execute either an $Batch_{odd}$ repair method or an $Batch_{even}$ repair method for the stripes in a $Batch_{single}$.

Staggering the upload and download traffic of two stripes rarely causes the upload and download traffic congestion. To implement this effect, whether it is an $Batch_{even}$ repair method or an $Batch_{odd}$ repair method, the idea of full duplex transmission is used to reduce the data transfer time. The cross-rack data transfer is divided into two parallel data transfer methods: forward data transfer and reverse data transfer. BPR lets Odd-numbered stripes implement forward parallel data transfer and even-numbered stripes implement reverse parallel data transfer.

The so-called forward parallel data transfer is to transfer data in parallel from the source racks with a larger number to the rack with a smaller number, i.e. $R4$ transfers data to $R3$ in Figure 6(a), and finally transfer the data to the destination rack $R1$. While the reverse parallel data transfer is the opposite, i.e., to transfer data in parallel from the source rack with a smaller number to the rack with a larger number, i.e. $R3$ transfers data to $R4$ in Figure 6(a), and finally transfer the data to the destination rack $R6$.

There is a slight difference between $Batch_{even}$ repair method and $Batch_{odd}$ repair method. Firstly, the process of forward parallel data transfer in $Batch_{even}$ repair method is introduced in detail. Forward parallel data transfer starts by grouping the source racks of the stripe in two-by-two order from the smallest number to the largest one. Then, the chunks in the rack with the larger number in the two racks of the

same group are aggregated and transferred to the rack with the smaller number, and then aggregated with the chunks in the rack with the smaller number. The racks continue to repeat the above process of grouping and aggregation in pairs, until the aggregated chunks of all source racks are transmitted to the source rack with the smallest number, and then aggregated and transmitted to the destination rack of the stripe. Decoding operation is performed to get the chunks that need to be recovered.

There are a little difference between the $Batch_{odd}$ repair method and the $Batch_{even}$ repair method. During data transmission, there has a single source rack cannot participate in the parallel transmission in the $Batch_{odd}$. And the chunks of the single source rack are directly aggregated at the beginning of the recovery and transferred to the destination rack. As shown in Figure 6(b), $R5$ is a single source rack that is picked out. When the forward/reverse data transfer starts, the chunks are directly aggregated and transferred to the destination racks of $R1$ and $R6$.

VI. EFFECT ANALYSIS OF BPR

In this section, we will analyze the effect of BPR in two aspects: cross-rack data transfer time and recovery throughput during data recovery. The data transfer time mainly analyzes the time of cross-rack transfer chunks to the destination rack waiting to participate in data recovery in different cases, and the recovery throughput mainly analyzes the network load occupied by the BPR.

Because $RS(6,3)$ and $RS(12,4)$ are exemplified in some latest research [14] [16] [18] as classical examples. Next, we take the above $RS(6,3)$ and $RS(12,4)$ multi-stripes repair in Figure 6(a) and Figure 6(b) to analyze the differences in cross-rack data transfer time and recovery throughput between BPR in recovering data in many different scenarios.

The process of data recovery consists of network data transfer and decoding, but in actual recovery, the decoding time is much smaller than the data transfer time, so we do not consider the decoding time in this approximate calculation. Data transfer requires intra-rack aggregation and then cross-rack transfer, and the time is T_{inner} and T_{cross} respectively, often $T_{cross} \approx T_{inner} * 10$. In summary, cross-rack transfer is the key to data recovery, so we focus on the time to cross-rack data transfer.

Because the full-duplex transmission can be sent (uploaded) and received (downloaded) independently at the same transmission rate due to the widespread support of network interface cards (NICs) and network cables, the time of cross-rack data transfer can be represented by the time to upload chunks T_{upload} and download chunks $T_{download}$, which means that

$$T_{cross} = T_{upload} = T_{download} \quad (2)$$

Thanks to the partial decoding technology, we can aggregate chunks in the rack into a chunk whose size is same as a normal chunk for transmission, we can set the time of one cross-rack transmission as $T_{oncecross}$. (n is the number of the cross-rack data transfer)

$$T_{cross} = n * T_{oncecross} \quad (3)$$

$$T_{oncecross} = T_{onceupload} = T_{oncedownload} = T \quad (4)$$

The time T for uploading/downloading a chunk is thus set as the unit time for cross-rack data transfer. Discuss cross-rack data transfer time when recovering data by scenario.

Case 1: RS(6,3) in Figure 6(a).

When we adopt BPR, the recovery time graph is Figure 7.

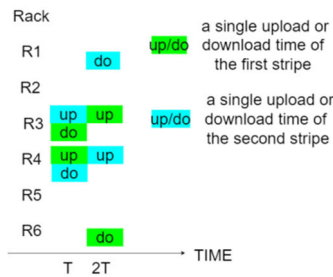


FIGURE 7. Time analyses of recovering two stripes in RS(6, 3) with BPR repair method.

From the graph in Figure 7, we can see that the time $T_{cross} = 2T$ for cross-rack data transfer in BPR. There is no upload and download traffic jam.

For RS(n, m), $k = \lceil \frac{n+m}{m} \rceil$. If the destination rack is a non-host rack, the cross-rack transfer time for 2 stripes recovery in Batch_{even} repair method is

$$T_{Batcheven} = \lceil (\log_2(k + 1)) \rceil \quad (5)$$

If the destination rack is the non-full rack, for the destination rack isn't involved in transferring chunks through cross-rack, the cross-rack transfer time for 2 stripes recovery in

Batch_{even} repair method is

$$T_{Batcheven} = \lceil (\log_2(k)) \rceil \quad (6)$$

Case 2: RS(12,4) in Figure 6(b).

When we adopt BPR, the recovery time graph is Figure 8.

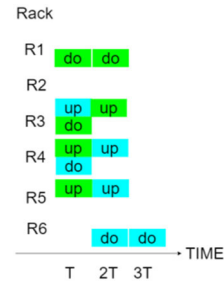


FIGURE 8. Time analyses of recovering two stripes in RS(12, 4) with BPR repair method.

From the graph in Figure 8, we can see that the time $T_{cross} = 3T$ for cross-rack data transfer in BPR. There is a little upload traffic jams in the R5 when it transfers chunks to different racks.

For RS(n, m), $k = \lceil \frac{n+m}{m} \rceil$, the cross-rack transfer time for 2 stripes recovery in Batch_{odd} repair method is

$$T_{Batchodd} = \lceil (\log_2(k + 1)) + 1 \rceil \quad (7)$$

Also, if the destination rack is the non-host rack, the cross-rack transfer time for 2 stripes recovery in Batch_{even} repair method is

$$T_{Batchodd} = \lceil (\log_2(k)) + 1 \rceil \quad (8)$$

We also evaluate the advantage of reducing upload and download traffic jams on the recovery throughput. Recovery throughput is defined as recovered data volume per second.

$$T_p = \frac{\text{Total data volume}}{\text{Total transfer time}} \quad (9)$$

VII. PERFORMANCE EVALUATION

In this section, we present extensive experiment results to evaluate the performance of recovery algorithm BPR.

System configuration: we conduct the experiments on a cluster of seven virtual machines. In this part, each virtual machine is created based on an Ubuntu 16.04.1 with 8 cores CPU, 5GB RAM, and 40GB SCSI. We use one virtual machine as a control terminal and use six virtual machines to simulate six racks. Each virtual machine has four processes to simulate four nodes. The bandwidth of the switch is 1Gbps.

We focus on the single failure scenario, in which a random data chunk in the encoded stripe in the layout of PDL is assumed to have failed. In our experiments, each chunk is configured to be 128MB. In this part, we compare the repair performance in two aspects, the cross-rack transfer time and the recovery throughput.

Because RS(6,3) and RS(12,4) are involved in the layout of PDL, and RS(6,3), RS(12,4) are applicable to Batch_{even}

repair method and $Batch_{odd}$ repair method, we first conduct three control trial for RS(6,3) and RS(12,4) separately to analyze the cross-rack transfer time in the different numbers of stripes. These stripes are all damaged stripes waiting for repair.

Figure 9 and Figure 10 shows the result, we can derive the following findings: First, the cross-rack transfer time with BPR is similar to the time with rPDL in minority stripes recovery (20/30 stripes), even the recovery time is a little longer. Because in the process of chunks recovery in the minority stripes, I/O and aggregation time account for a large proportion with the partial decoding technique. And upload and download traffic jams caused by the minority stripes recovery is not many.

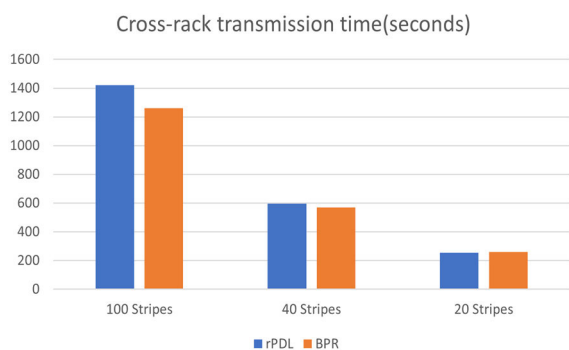


FIGURE 9. Total cross-rack transfer time for rPDL and BPR repair of single-chunk failures with the different numbers of stripes for RS(6, 3).

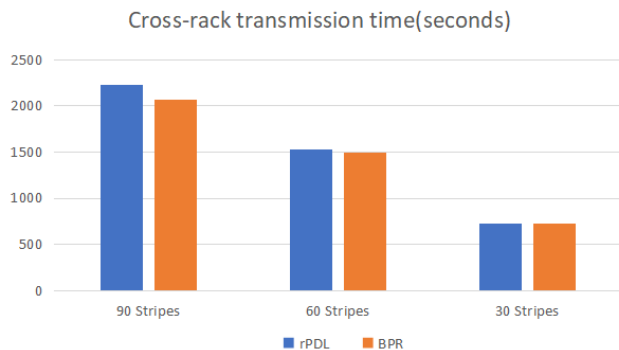


FIGURE 10. Total cross-rack transfer time for rPDL and BPR repair of single-chunk failures with the different numbers of stripes for RS(12, 4).

Then, we find that with the increase of the number of stripes, the improvement effect is getting better and better with BPR than rPDL. When recovery many stripes (100/90 stripes) BPR can reduce the cross-rack transfer time by 7%-10% when compared with rPDL, respectively. Next, we measure recovery throughput with 90 and 100 stripes in BPR_{odd} and BPR_{even} . And find that BPR increases the recovery throughput compared with the rPDL by 7-8% in Figure 11 which similar to the cross-rack transfer time for the reason that when k is not much, the total data volume of BPR

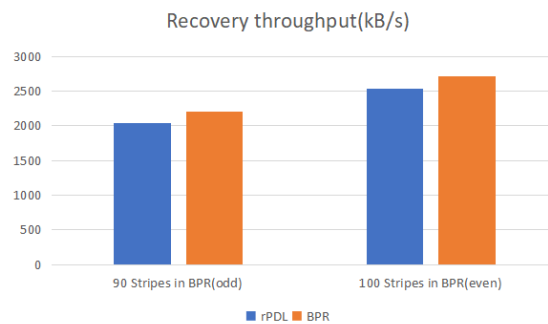


FIGURE 11. Recovery throughput for rPDL and BPR with the different numbers of stripes in BPR_{odd} and BPR_{even} .

is similar to rPDL and there is a negative correlation between recovery throughput with total transfer time.

VIII. CONCLUSION

In this paper, we focus on the cross-rack data transmission issue with respect to upload and download traffic jams when recover data chunks in Batches. We propose BPR, an Erasure Coding repair approach for the distributed storage system. BPR provides a stripe classification scheme and then constructs multi-stripe repair solutions based on the classification results. Through forward and reverse parallel transfer, BPR scatters and balances the traffic of upload and download. The experimental results show that BPR significantly reduces cross-rack transmission time by up to 10% and increases the recovery throughput by up to 8% compared with rPDL.

REFERENCES

- [1] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, Jun. 1960.
- [2] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM SIGCOMM Conf. SIGCOMM*, Aug. 2013, pp. 231–242.
- [3] Y. Hu, X. Li, M. Zhang, P. C. Lee, X. Zhang, P. Zhou, and D. Feng, "Optimal repair layering for erasure-coded data centers: From theory to practice," *ACM Trans. Storage*, vol. 13, no. 4, p. 33, 2017.
- [4] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. 10th ACM SIGCOMM Conf. Internet Meas.*, Nov. 2010, pp. 267–280.
- [5] R. Li, Y. Hu, and P. P. C. Lee, "Enabling efficient and reliable transition from replication to erasure coding for clustered file systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 9, pp. 2500–2513, Sep. 2017.
- [6] Z. Shen, J. Shu, and P. P. C. Lee, "Reconsidering single failure recovery in clustered file systems," in *Proc. 46th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2016, pp. 323–334.
- [7] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multiresource fairness for correlated and elastic demands," in *Proc. 13th USENIX Conf. Netw. Syst. Des. Implement.*, 2016, pp. 407–424.
- [8] H. Liu, M. K. Mukerjee, C. Li, N. Feltman, G. Papen, S. Savage, S. Seshan, G. M. Voelker, D. G. Andersen, M. Kaminsky, G. Porter, and A. C. Snoeren, "Scheduling techniques for hybrid circuit/packet networks," in *Proc. 11th ACM Conf. Emerg. Netw. Exp. Technol.*, Dec. 2015, p. 41.
- [9] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," *Very Large Data Based Endowment*, vol. 6, no. 5, pp. 325–336, Mar. 2013.

- [10] Y. Hu, Q. Li, W. Xie, and Z. Ye, "An ant colony optimization based data update scheme for distributed erasure-coded storage systems," *IEEE Access*, vol. 8, pp. 118696–118706, 2020, doi: [10.1109/ACCESS.2020.3004577](https://doi.org/10.1109/ACCESS.2020.3004577).
- [11] X. Y. Zhang and Y. C. Hu, "Efficient storage scaling for MBR and MSR codes," *IEEE Access*, vol. 8, pp. 78992–79002, 2020, doi: [10.1109/ACCESS.2020.2989822](https://doi.org/10.1109/ACCESS.2020.2989822).
- [12] Z. Shen, S. Lin, J. Shu, C. Xie, Z. Huang, and Y. Fu, "Cluster-aware scattered repair in erasure-coded storage: Design and analysis," *IEEE Trans. Comput.*, vol. 70, no. 11, pp. 1861–1874, Nov. 2021.
- [13] H. Shi and X. Lu, "INEC: Fast and coherent in-network erasure coding," in *Proc. SC20, Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2020, pp. 1–17, doi: [10.1109/SC41405.2020.00070](https://doi.org/10.1109/SC41405.2020.00070).
- [14] L. Xu, M. Lyu, Z. Li, C. Li, and Y. Xu, "A data layout and fast failure recovery scheme for distributed storage systems with mixed erasure codes," *IEEE Trans. Comput.*, vol. 71, no. 8, pp. 1740–1754, Aug. 2021.
- [15] Y. Bai, Z. Xu, H. Wang, and D. Wang, "Fast recovery techniques for erasure-coded clusters in non-uniform traffic network," in *Proc. 48th Int. Conf. Parallel Process.*, Aug. 2019, pp. 1–10.
- [16] H. Zhou, D. Feng, and Y. Hu, "Bandwidth-aware scheduling repair techniques in erasure-coded clusters: Design and analysis," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 12, pp. 3333–3348, Dec. 2022, doi: [10.1109/TPDS.2022.3153061](https://doi.org/10.1109/TPDS.2022.3153061).
- [17] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Conf. Operating Syst. Design Implement. (USA)*, 2010, pp. 61–74.
- [18] T. Liu, S. Alibhai, and X. He, "A rack-aware pipeline repair scheme for erasure-coded distributed storage systems," in *Proc. 49th Int. Conf. Parallel Process. (ICPP)*, Aug. 2020, pp. 1–11.
- [19] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, pp. 81–94.
- [20] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2012, pp. 15–26.
- [21] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's warm BLOB storage system," in *Proc. USENIX Symp. Oper. Syst. Design Implement.*, 2014, pp. 383–398.



YING SONG received the Ph.D. degree in computer engineering from the Institute of Computing Technology (ICT), Chinese Academy of Sciences. She is currently an Associate Professor with the Computer School, Beijing Information Science and Technology University. Her work has covered topics, such as performance modeling, resource management, cloud computing, and big data computing platform. She has been authored or coauthored more than 30 publications in these areas, since 2007. Her main research interests include computer architecture, parallel and distributed computing, and virtualization technology. She served in various academic conferences.



WENXUAN ZHAO received the B.S. degree from Guangzhou University, Guangdong, China, in 2021. He is currently pursuing the master's degree with the Computer School, Beijing Information Science and Technology University, Beijing, China. His main research interest includes distributed storage.



BO WANG received the B.S. degree in computer science from Northeast Forestry University (NEFU), Harbin, China, in 2010, and the Ph.D. degree in computer science from Xi'an Jiaotong University (XJTU), Xi'an, China, in 2017. He was a Guest Student with the State Key Laboratory of Computer Architecture, Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), from 2012 to 2016. He is currently a Lecturer with the Software Engineering College, Zhengzhou University of Light Industry (ZZULI). His research interests include distributed systems, cloud computing, edge computing, resource management, and task scheduling. He has published more than ten research articles in these areas. He served in various academic journals and conferences.

• • •