

Cluster-Aware Scattered Repair in Erasure-Coded Storage: Design and Analysis

Zhirong Shen¹, Member, IEEE, Shiyao Lin, Jiwu Shu, Fellow, IEEE, Chengxin Xie, Zhijie Huang, and Yingxun Fu², Member, IEEE

Abstract—Erasure coding is a storage-efficient means to guarantee data reliability in today's commodity storage systems, yet its repair performance is seriously hindered by the substantial repair traffic. Repair in clustered storage systems is even complicated because of the scarcity of the cross-cluster bandwidth. We present ClusterSR, a cluster-aware scattered repair approach. ClusterSR minimizes the cross-cluster repair traffic by carefully choosing the clusters for reading and repairing chunks. It further balances the cross-cluster repair traffic by scheduling the repair of multiple chunks. Large-scale simulation and Alibaba Cloud ECS experiments show that ClusterSR can reduce 5.6–52.7 percent of the cross-cluster repair traffic and improve 14.4–68.8 percent of the repair throughput.

Index Terms—Cross-cluster repair traffic, scattered repair, load balancing, full duplex transmission

1 INTRODUCTION

LARGE-SCALE clustered storage systems, often built on hundreds or even thousands of storage servers (also called *nodes*), have to tackle prevalent unexpected failures [19]. To guarantee data reliability against failures, pre-storing additional data redundancy is a commonly adopted approach in production systems [2], [11], [26], [27], [28], where replication and erasure coding are two representatives. Compared to replication, erasure coding [2], [18], [26], [28] is much more storage-efficient, which can attain the same degree of fault tolerance with far less storage redundancy [38]. Generally, erasure coding takes pieces of fixed-size data information (called *data chunks*) as input and generates a small number of equal-size redundant chunks (called *parity chunks*) through a predefined encoding functionality. If any data or parity chunk accidentally fails, erasure coding can retrieve a subset of the surviving data and parity chunks to restore the original chunk. Because of its high storage efficiency, erasure coding is more preferable in today's production systems, such as Hadoop HDFS [3], Windows Azure Storage [17], and Facebook f4 [26].

While being storage-efficient, erasure coding incurs substantial *repair traffic* (i.e., data retrieved for repair). For example, Reed-Solomon codes (RS codes) [31], which are a well-known family of erasure codes, demand to retrieve the chunks whose size may be even several times that of the lost data for repair (Section 2.2). Repair becomes more complicated in large-scale clustered storage systems. Modern clustered storage systems usually organize nodes into multiple clusters in a hierarchical manner, where nodes are first grouped into a *cluster* connected via a common switch and the switches are then interconnected through the network core [6], [10], [16], [33]. In such network architecture, the cross-cluster bandwidth, which is competed among the nodes within the same cluster for various workloads (e.g., replication writes [6] and shuffle in Map-Reduce jobs [4]), is often oversubscribed and shown to be much more scarce than the intra-cluster bandwidth (Section 2.1). Hence, the repair that incurs heavy *cross-cluster repair traffic* (i.e., data retrieved across clusters for repair) will significantly prolong the repair process and take more repair time.

To alleviate the influence of the cross-cluster repair traffic, existing studies design new families of cluster-aware erasure codes [15], [16] to sustain the same fault tolerance degree with less cross-cluster repair traffic, or develop new repair scheduling approach to minimize the cross-cluster repair traffic [36]. However, these prior designs all consider the *dedicated repair* scenario, which repairs all the failed chunks in a dedicated node. Such repair scenario easily makes the bandwidth of the dedicated node be the performance bottleneck of the repair.

In this paper, we strive to remove this performance bottleneck and reconsider the repair in erasure-coded clustered storage. We mainly focus on the *scattered repair* scenario, which stores the repaired chunks across all the surviving nodes in the clustered storage. Our observations are two-fold. On one hand, as the cross-cluster bandwidth seriously hinders the repair procedure, it becomes crucial to minimize

- Zhirong Shen is with the School of Informatics, Xiamen University, Xiamen, Fujian 361005, China, and also with the State Key Laboratory of Integrated Services Networks, Xidian University, Xian, Shaanxi 710126, China. E-mail: shenzr@xmu.edu.cn.
- Shiyao Lin and Chengxin Xie are with the School of Informatics, Xiamen University, Xiamen, Fujian 361005, China. E-mail: linesyoo@gmail.com, 281986011@qq.com.
- Jiwu Shu is with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: shujw@tsinghua.edu.cn.
- Zhijie Huang is with the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019 USA. E-mail: zhijie.huang@uta.edu.
- Yingxun Fu is with the School of Computer Science, North China University of Technology, Beijing 100144, China. E-mail: mooncape@hotmail.com.

Manuscript received 29 May 2020; revised 26 Sept. 2020; accepted 28 Sept. 2020. Date of publication 1 Oct. 2020; date of current version 8 Oct. 2021. (Corresponding author: Jiwu Shu.)

Recommended for acceptance by P. Stenstrom.

Digital Object Identifier no. 10.1109/TC.2020.3028353

the cross-cluster repair traffic in scattered repair as well. On the other hand, as network interface cards (NICs) and network cables extensively support *full duplex* transmission [7], [23], which can send (upload) and receive (download) data independently at the same transmission rate, balancing the *cross-cluster upload and download traffics* (i.e., data uploaded and downloaded across clusters) for repair is essential to further reduce the repair time.

Transplanting existing scattered repair approaches [34] in data centers still faces several non-trivial challenges. The first is that conventional scattered repair approaches neglect the *bandwidth diversity* phenomenon (i.e., the cross-cluster bandwidth is much scarcer). Directly deploying them in data centers cannot minimize the cross-cluster repair traffic and hence easily prolongs the repair process. The second is that conventional scattered repair approaches ignore the reality of extensive full duplex transmission supported in today's NICs and network cables. Therefore, they cannot well balance the upload and download repair traffic, hence falling short of further shortening the repair procedure.

We therefore present **ClusterSR**, a *Cluster-aware Scattered Repair* approach that aims to minimize and balance the cross-cluster repair traffic. **ClusterSR** first carefully examines the data distribution and determines the *repair solution* (which specifies the nodes to read the surviving data and store the repaired data) for each failed chunk, with the primary objective of minimizing the cross-cluster repair traffic. It then seeks to schedule the repair of multiple chunks, such that the resulting cross-cluster upload and download traffics are both balanced across clusters. To our best knowledge, **ClusterSR** is the *first* work that considers minimizing and balancing both of the cross-cluster upload and download traffics in scattered repair.

In summary, we make the following contributions.

- 1) We formulate the problem of cluster-aware scattered repair in erasure-coded clustered storage and identify that the lower bound of the repair time can be attained by minimizing and balancing the cross-cluster upload and download traffics.
- 2) We present **ClusterSR**, a cluster-aware scattered repair approach. **ClusterSR** carefully chooses the nodes that participate in a single chunk's repair to minimize the cross-cluster repair traffic. It additionally seeks to schedule the repair of multiple chunks for balancing the cross-cluster upload and download traffics. **ClusterSR** is a general design for different erasure codes.
- 3) We implement a **ClusterSR** prototype in C++ and show that it can be effortlessly tuned for assisting the repair in HDFS of Hadoop 3.1.1 [3].
- 4) We evaluate **ClusterSR** via large-scale simulation and Alibaba Cloud Elastic Compute Service (ECS) [1] experiments to demonstrate its scalability and effectiveness in real-world environments. We show that **ClusterSR** can reduce 8.6-52.7 percent of the cross-cluster repair traffic and improve 14.4-68.8 percent of the repair throughput. We also demonstrate that **ClusterSR** is effective on balancing the cross-cluster upload and download traffics.

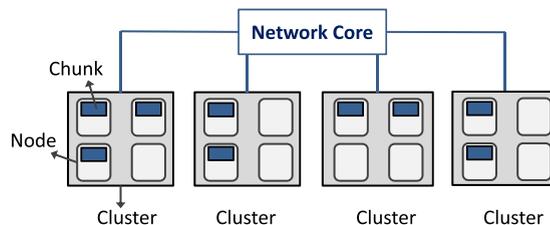


Fig. 1. Example of a clustered storage system deployed with RS(9,6).

The source code of **ClusterSR** can be reached via: <https://github.com/shenzr/clustersr>

2 BACKGROUND

2.1 Clustered Storage

We consider the clustered storage with a two-level hierarchical architecture, in which nodes are first organized into *clusters* and multiple clusters are then interconnected via the network core. A cluster can physically be a rack [33], [36] or even a data center [5]. Fig. 1 depicts the architecture of the clustered storage. Such architecture has been applied in modern data center deployment [10], [26] and assumed in previous work [6], [16], [33], [36].

The hierarchical architecture results in the *bandwidth diversity* phenomenon, where the cross-cluster bandwidth is often oversubscribed [4], [6], [13] and therefore appears more scarce than the intra-cluster bandwidth. To define the scarcity of the cross-cluster bandwidth, previous studies use the *oversubscription ratio* calculated as the ratio of the intra-cluster bandwidth and the cross-cluster bandwidth. They find that the oversubscription ratio normally varies from 5 to 20 [4], [6], and even reaches 240 in some extreme cases [13].

2.2 Erasure Coding

Erasure coding often operates on *chunks*, which are a collection of fixed-size information in units of MBs (e.g., 64 MB by default in Hadoop HDFS [3]). In this paper, we mainly focus on the linear codes, including RS codes [31], regenerating codes [9], [29], and locally repairable codes (LRCs) [17], [32], [37]. For easy presentation, we mainly use RS codes as an instance to clarify our algorithmic designs. We also show that **ClusterSR** can be readily extended for regenerating codes and LRCs (Section 4.1).

RS codes often use two parameters, namely k and n (where $k < n$), to configure their storage efficiency and fault-tolerance capability, which can be denoted by $RS(n, k)$. In the encoding stage, $RS(n, k)$ takes k data chunks as input and generates additional $n - k$ parity chunks via linear arithmetics over Galois finite field [31]. These n data and parity chunks that are generated together in the encoding stage collectively constitute a *stripe*, such that any k chunks of a stripe can decode (recover) the original k data chunks; in other words, $RS(n, k)$ can tolerate *any* $n - k$ chunk failures within a stripe. In the following discussion, we use the term “chunks” to refer to the data and parity chunks for brevity, as all of them are treated equally in the repair.

Therefore, by distributing the n chunks of each stripe across n distinct nodes (i.e., one chunk per node), $RS(n, k)$ can tolerate *any* $n - k$ node failures. Besides, if a cluster is

allowed to store at most $n - k$ chunks of each stripe, then we can attain *cluster-level fault tolerance* (i.e., tolerating any single cluster failure), as we can always find at least k chunks of the same stripe for repair from other clusters. Fig. 1 shows that the nine chunks of a stripe encoded by RS(9,6) (i.e., $n = 9$ and $k = 6$) are stored in a clustered storage system with four clusters, where each cluster stores at most $n - k$ chunks (i.e., three in this example). Such a deployment can tolerate any single cluster failure.

2.3 Repair in Erasure Coding

Repairing in erasure coding is an I/O intensive operation. For instance, RS(n, k) requires retrieving k surviving chunks to repair a chunk, indicating that the storage and network I/Os for repair are k times the size of the failed chunk. To improve the repair efficiency, regenerating codes [9], [29] trade additional computation cycles for reduced repair traffic. To repair a chunk, the surviving node will send a subchunk computed as a linear combination of the locally stored data. These subchunks are then assembled to restore the failed chunk. To further reduce the amount of storage I/O incurred in repair, recently proposed regenerating codes [29] obviate the need of linear computations performed on the surviving nodes, meaning that the subchunks can be directly read from the local storage for repair.

On the other hand, LRCs [17], [32], [37] save repair traffic by maintaining slightly more parity chunks. They categorize the k data chunks of a stripe into several *local groups*, and generate a local parity chunk based on the data chunks of the same local group. Hence, LRCs can repair a chunk by merely retrieving the surviving chunks of the same local group.

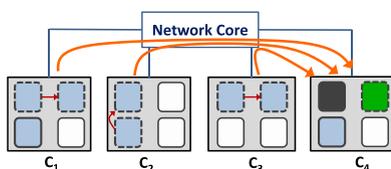
3 OBSERVATIONS AND PROBLEM FORMULATION

3.1 Observations

Given the scarcity of the cross-bandwidth, we have the following two observations. To clarify, we use the clustered storage system in Fig. 1 as an instance and label the four clusters by $\{C_1, C_2, C_3, C_4\}$.

Observation 1. We first notice that the nodes selected in repair directly determine the cross-cluster repair traffic. Based on the data layout in Fig. 1, suppose that a node in the cluster C_4 fails and the system chooses a *destination node* in C_4 to store the repaired chunk. As the system in Fig. 1 uses RS(9,6), it requires to retrieve any six surviving chunks to perform repair.

Fig. 2 shows two repair solutions with different cross-cluster repair traffics, where the chunks selected for repaired are marked in dash lines. Specifically, the first repair solution (Fig. 2a) retrieves six surviving chunks from $\{C_1, C_2, C_3\}$ and stores the repaired chunk in C_4 . By relying



(a) A repair solution that transmits three chunks across clusters.

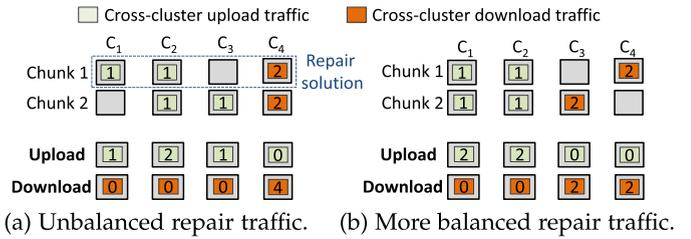


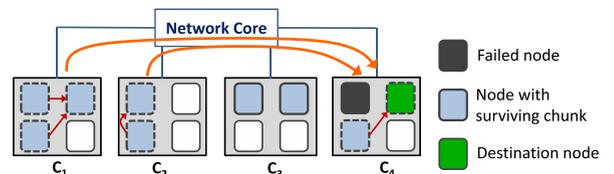
Fig. 3. Observation 2: Unbalanced cross-cluster upload and download traffics prolong the repair procedure.

on the linearity of the repair (decoding) operation, a cluster that has chunks requested can aggregate these chunks into an aggregated chunk (whose size is the same as the original data chunk [36]) and therefore only needs to send one chunk to C_4 [36]. Consequently, Fig. 2a transmits three chunks across clusters for repair. As a comparison, the repair solution in Fig. 2b reads six surviving chunks from $\{C_1, C_2, C_4\}$. As the chunk requested in C_4 can be directly transmitted within the same cluster, Fig. 2b merely needs to send two chunks across cluster to accomplish the repair.

Observation 2. We also identify that when repairing multiple chunks, the unbalanced repair solutions easily prolong the repair procedure. Fig. 3 presents an example that illustrates the cross-cluster upload and download traffics of repairing two chunks. For example, the repair solution of the first chunk in Fig. 3 is based on Fig. 2b, where the two clusters $\{C_1, C_2\}$ send (upload) one chunk across clusters, while the cluster C_4 receives (download) two chunks. We can also notice that for a repair solution, the induced cross-cluster upload traffic is equal to the cross-cluster download traffic. Figs. 3a and 3b have two different options in repairing the second chunk, and hence result in different distributions of cross-cluster upload and download traffics. For example, C_4 is the most loaded cluster in Fig. 3a and needs to download four chunks from other clusters. By contrast, each of the four clusters in Fig. 3b has to send or receive two chunks across clusters. As NICs and network cables extensively support full duplex technology (i.e., a node can send and receive data independently at the same transmission rate), the repair procedure is bottlenecked by the cluster that affords the maximum cross-cluster upload or download traffic. Therefore, a repair solution with balanced cross-cluster upload and download traffics can well shorten the repair process.

3.2 Modeling

We further formulate the repair problem based on the following assumptions. First, the computation time in repair is often trivial [19], [36] and can be negligible. Second, as each node can be attached with multiple disks, the cumulative



(b) A repair solution that transmits two chunks across clusters.

Fig. 2. Observation 1: The selection of nodes for repair (marked in dash lines) directly determines the cross-cluster repair traffic.

disk I/O bandwidth of a node is much larger than its NIC speed [6], making the network transmission be the true bottleneck in repair. Third, because of the scarcity of the cross-cluster bandwidth [4], [6], [13], we assume that the cross-cluster transmission dominates the network transmission. Fourth, we focus on single failure, which accounts for more than 90 percent of all the failure events in practical storage deployment [19].

Suppose that the clustered storage system consists of l clusters termed $\{C_1, C_2, \dots, C_l\}$ and the capacity of the cross-cluster bandwidth assigned for repair is b . Let u_i and d_i be the amounts of the cross-cluster upload and download traffics for repair over the i th cluster (where $1 \leq i \leq l$), respectively. Then the most cross-cluster upload and download traffics can be denoted by \bar{u} and \bar{d} , where $\bar{u} = \max\{u_i | 1 \leq i \leq l\}$ and $\bar{d} = \max\{d_i | 1 \leq i \leq l\}$, respectively. Obviously, the repair time is determined by $m = \max\{\bar{u}, \bar{d}\}$ and can be given by $T = \frac{m}{b}$. As the cross-cluster upload and download traffics have equal size (i.e., $\sum_{i=1}^l u_i = \sum_{i=1}^l d_i$), the average cross-cluster upload and download traffics loaded on a cluster are equal and can both be calculated by $a = \frac{1}{l} \sum_{i=1}^l u_i$. Hence, we can have

$$T = \frac{m}{b} \geq \frac{a}{b}. \quad (1)$$

The equation holds if $m = a = \frac{1}{l} \sum_{i=1}^l u_i$, implying that the cross-cluster upload and download traffics are both evenly distributed across the l clusters.

We can further derive the lower bound of the repair time. Suppose that a^* is the lower bound of a . Finally, based on Equation (1), we can have

$$T = \frac{m}{b} \geq \frac{a}{b} \geq \frac{a^*}{b}. \quad (2)$$

Equation (2) indicates that the minimum repair time can be achieved when the cross-cluster upload and download repair traffics are both balanced (i.e., $m = a$) and minimized (i.e., $a = a^*$).

3.3 Objective Formulation

To attain the minimum repair time, our objective is to make the cross-cluster upload and download traffics most balanced, with the constraint that their amounts have been minimized. This objective can be formulated as follows:

$$\begin{aligned} & \text{minimize} && \frac{m}{a} \\ & \text{subject to} && a = a^*. \end{aligned}$$

We call the objective function $\frac{m}{a}$ the *load balancing rate*. Therefore, the minimum load balancing rate is one (i.e., when $m = a$), when the cross-cluster upload and download traffics for repair are both evenly distributed across the l clusters.

4 CLUSTER-AWARE SCATTERED REPAIR

We now present ClusterSR, a cluster-aware scattered repair approach. ClusterSR is composed of two components. The first is to find the repair solutions (which specify the nodes

for reading data and performing repair) with the minimum cross-cluster repair traffic for each stripe (Section 4.1). The second is a greedy algorithm that seeks to schedule the repair of multiple chunks and searches their repair solutions, such that the cross-cluster upload and download traffics are both balanced across clusters (Section 4.2). We also analyze the computational complexity of ClusterSR (Section 4.3), and finally show via rigorous theoretical analysis to demonstrate the reliability improvement gained by ClusterSR (Section 4.4).

4.1 Minimizing Cross-Cluster Repair Traffic

We first consider the minimization of the cross-cluster repair traffic in scattered repair. The *main idea* behind ClusterSR is to access the fewest clusters for collecting sufficient surviving chunks, and choose a destination node without violating the cluster-level fault tolerance. ClusterSR then performs the intra-cluster aggregation on the requested chunks within the same cluster, such that each accessed cluster can merely send one aggregated chunk to the destination node for repair. Algorithm 1 elaborates the detail procedures.

Algorithm 1. Minimizing Cross-Cluster Repair Traffic

Input: A failed stripe

Output: A valid repair solution for the failed stripe

- 1: // Find the fewest clusters for retrieving data
 - 2: Get $\{C'_1, C'_2, \dots, C'_l\}$, where $h_1 \geq h_2 \geq \dots \geq h_l$
 - 3: Establish a smallest value v , such that $\sum_{1 \leq i \leq v} h_i \geq k$
 - 4: // Find a destination cluster
 - 5: **for** $1 \leq i \leq l$ **do**
 - 6: **if** $h_i < n - k$ **then**
 - 7: $C^* = C'_i$
 - 8: **break**
 - 9: **end if**
 - 10: **end for**
 - 11: // Perform repair
 - 12: Select k surviving chunks from $\{C'_i | 1 \leq i \leq v\} \cup C^*$
 - 13: Select a destination node from C^*
 - 14: **for** $1 \leq i \leq v$ **do**
 - 15: Aggregate the selected chunks in C'_i
 - 16: Send the aggregated chunk to the destination node
 - 17: **end for**
 - 18: Perform repair and restore the failed chunk
-

Algorithm Details. Once identifying a failed stripe with a chunk loss, ClusterSR first sorts the clusters based on the number of surviving chunks of the failed stripe in descending order, and get the sorted clusters $\{C'_1, C'_2, \dots, C'_l\}$, where h_i is the surviving chunks of C'_i ($1 \leq i \leq l$). It then establishes a smallest value v (where $1 \leq v \leq l$), such that the first v clusters after sorting have at least k surviving chunks for repair (Lines 2-3).

To select the cluster for storing the repaired chunk, ClusterSR scans the sorted clusters and finds the first one (denoted by C^*) that has less than $n - k$ surviving chunks of the failed stripe (Lines 4-10). We call C^* the *destination cluster*. The selection of C^* ensures that the cluster-level fault tolerance can still be guaranteed even after repair (i.e., C^* still stores no more than $n - k$ chunks of the failed stripe after repair). ClusterSR then chooses k surviving chunks

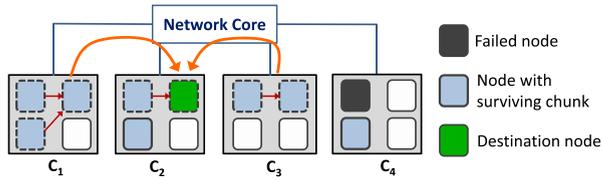


Fig. 4. A repair solution with the least repair traffic for RS(9, 6).

from the union of the first v sorted clusters and C^* (Line 12). It also picks a node from C^* to serve as the destination node, with the requirement that the destination node should not store any chunk of the failed stripe before repair (Line 13). Finally, for each of the v clusters, ClusterSR aggregates the requested chunks and transmits the aggregated chunk to C^* for repair (Lines 14-18).

Example. We show an example via Fig. 4 to clarify the repair process of Algorithm 1. Suppose that the system deploys RS(9,6) (i.e., $n = 9$ and $k = 6$) and a node in C_4 fails at this time. Then we can obtain the sorted clusters $\{C_1, C_2, C_3, C_4\}$ based on the number of surviving chunks they have. We can deduce that $v = 3$, as the first three clusters have seven chunks, whose number is no smaller than k . We then find that C_2 can serve as the destination cluster, as it is the first cluster among the sorted ones that have less than $n - k = 3$ chunks. We choose a destination node in C_2 that does not store any chunk of the failed stripe before repair. Finally, we select six chunks (marked in dashed lines) from the first three clusters, aggregate the selected chunks for each cluster, and transmit the aggregated chunks to the destination node in C_2 . Thus, we only transmit two chunks across clusters for repair.

Discussion. Algorithm 1 is designed to capture one of the repair solutions that can attain the least cross-cluster repair traffic for a failed stripe. For example, in Fig. 4, we can change the destination cluster to be C_3 , and selectively read six chunks from $\{C_1, C_2, C_3\}$. This repair solution also only needs to transmit two chunks across clusters after aggregation (from C_1 and C_2 to C_3). Therefore, we say a repair solution is *valid* for a failed stripe if it can repair the failed chunk with the least cross-cluster repair traffic [36]. All valid repair solutions of a stripe will be considered when trying to balance the cross-cluster repair traffic (Section 4.2).

Optimality. We now prove that the repair solution found by Algorithm 1 incurs the least cross-cluster repair traffic for RS(n, k), without violating the cluster-level fault tolerance. We can readily deduce that v is the smallest number of clusters that have at least k available chunks to repair the failed chunk by contradiction (if we can find $v' < v$ clusters to collect at least k surviving chunks, then it violates the assumption that v is the smallest value that stores at least k surviving chunks).

After establishing the v clusters for data retrieval, the selection of the destination cluster C^* directly determines the cross-cluster repair traffic. There are two possibilities in selecting C^* . If each C'_i in $\{C'_1, C'_2, \dots, C'_v\}$ has exactly $n - k$ surviving chunks of the failed stripe, then $C^* \notin \{C'_1, C'_2, \dots, C'_v\}$ and the minimum number of chunks transmitted across clusters is v . Otherwise, C^* will store more than $n - k$ chunks of the failed stripe after repair, thereby violating the cluster-level fault tolerance (each cluster is allowed to store at most $n - k$ chunks for promising cluster-level fault tolerance).

On the other hand, if some cluster C_i in $\{C'_1, C'_2, \dots, C'_v\}$ stores fewer than $n - k$ surviving chunks of the failed stripe, then we have $C^* \in \{C'_1, C'_2, \dots, C'_v\}$. In this case, each C_i in the first v clusters (except C^*) will send an aggregated chunk to C^* and hence the number of chunks transmitted across clusters is $v - 1$. We can prove that $v - 1$ is the minimum value via contradiction. We assume that C^* can repair the failed chunk by reading chunks from another v' clusters (where $v' < v - 1$). That said, we can find $v' + 1 < v$ clusters (i.e., the v' clusters plus C^*) for retrieving k surviving chunks. This violates the condition that v is the smallest number of clusters that have sufficient surviving chunks for repair.

Algorithm 2. Balancing Cross-Cluster Repair Traffic

Input: \mathcal{U} (The residual chunks), r (number of chunks repaired in a repair round), and t (number of steps)

Output: Chunks to be repaired in each repair round

```

1: function Repair $\mathcal{U}$ 
2:   // Initialize a set of chunks to be repaired
3:   Set  $\mathcal{R} \subset \mathcal{U}$ 
4:   Construct  $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \dots, \mathcal{S}_{|\mathcal{R}|}\}$  for  $\mathcal{R}$ 
5:   // Balance the cross-cluster repair traffic
6:   while true do
7:     if Substitute $\mathcal{R}, \mathcal{S}$  equals False then
8:       Swap $\mathcal{R}, \mathcal{S}, \mathcal{U}$ 
9:     end if
10:    Set  $t = t - 1$ 
11:    if  $t = 0$  then
12:      break
13:    end if
14:  end while
15:  Set  $\mathcal{U} = \mathcal{U} - \mathcal{R}$ 
16:  return ( $\mathcal{R}, \mathcal{S}, \mathcal{U}$ )
17: end function
18: procedure Main( $\mathcal{U}$ )
19:   Initialize  $e = 0$ 
20:   while  $\mathcal{U} \neq \emptyset$  do
21:     Set  $e = e + 1$ 
22:     ( $\mathcal{R}_e, \mathcal{S}_e, \mathcal{U}$ ) = Repair( $\mathcal{U}$ )
23:   end while
24:   return ( $\{\mathcal{R}_1, \mathcal{S}_1\}, \dots, \{\mathcal{R}_e, \mathcal{S}_e\}$ )
25: end procedure

```

Extension. Though Algorithm 1 mainly focuses on RS codes, it can be effortlessly tuned for LRCs and regenerating codes (Section 2.3). Due to page limits, we only use LRCs as an instance. For LRCs, to repair a global parity chunk, we can sort the clusters based on the number of surviving chunks of the k data chunks, and choose the fewest v clusters that have enough surviving chunks for repair. When selecting the destination cluster C^* in LRCs, one should ensure that C^* 's failure is still an information-theoretically decodable pattern [17] after repair (i.e., the remaining parity chunks that can take effect in repair are no less than the failed chunks), such that the cluster-level fault tolerance is still preserved after repair. In addition, for the LRC that keeps only one local parity chunk in each local group, ClusterSR cannot save the cross-cluster repair traffic when repairing any single data chunk, as all the surviving chunks of the same local group are required in repair. However, we argue that ClusterSR can still retain its

effect for the LRCs [39] that store multiple local parity chunks within a local group.

4.2 Balancing Cross-Cluster Repair Traffic

After establishing the repair solution with minimized cross-cluster repair traffic for each failed stripe, we next consider the balancing of cross-cluster upload and download traffics in the repair of multiple chunks. For easy manipulation, we propose to partition the repair into multiple *repair rounds* that are iteratively performed, where each repair round will selectively repair a constant number of chunks (denoted by r). We then design Algorithms 2, 3, and 4, whose objectives are to seek a combination of r failed chunks as well as their repair solutions, such that the induced cross-cluster upload and download traffics in this repair round are balanced across clusters. Algorithm 2 is the main algorithm, whose main idea is to *iteratively* mitigate the cross-cluster traffic on the most loaded cluster via *substituting* the selected repair solutions with its alternatives and *swapping* the chunks to be repaired in a repair round. Algorithms 3 and 4 elaborate the substitution and swapping procedures, respectively.

Algorithm 3. Substitute Function

```

1: function Substitute( $\mathcal{R}$ ,  $\mathbb{S}$ )
2:   Derive  $\bar{u}$  and  $\bar{d}$  from  $\mathbb{S}$ 
3:   if  $\bar{d} > \bar{u}$  then
4:     Get  $C_x$ , where  $d_x = \bar{d}$ 
5:     for each solution  $\mathcal{S}_i \in \mathbb{S}$  do
6:       if  $\mathcal{S}_i$  performs repair in  $C_x$  then
7:         Find  $\mathcal{S}'_i$  by substituting  $C_x$  in  $\mathcal{S}_i$  with another destination cluster
8:         Set  $\mathbb{S}_i = \mathbb{S} - \mathcal{S}_i \cup \mathcal{S}'_i$ , and get  $\bar{d}_i$  from  $\mathbb{S}_i$ 
9:       end if
10:    end for
11:    Set  $i^* = \text{argmin}_i \{\bar{d}_i\}$ 
12:    if  $\bar{d}_{i^*} < \bar{d}$  then
13:      Set  $\mathbb{S} = \mathbb{S} - \mathcal{S}_{i^*} \cup \mathcal{S}'_{i^*}$ 
14:      return True
15:    else
16:      return False
17:    end if
18:  else
19:    Get  $C_x$ , where  $u_x = \bar{u}$ 
20:    for each solution  $\mathcal{S}_i \in \mathbb{S}$  do
21:      if  $\mathcal{S}_i$  reads data from  $C_x$  then
22:        Find  $\mathcal{S}'_i$  by substituting  $C_x$  in  $\mathcal{S}_i$  with another cluster for reading data
23:        Set  $\mathbb{S}_i = \mathbb{S} - \mathcal{S}_i \cup \mathcal{S}'_i$ , and get  $\bar{u}_i$  from  $\mathbb{S}_i$ 
24:      end if
25:    end for
26:    Set  $i^* = \text{argmin}_i \{\bar{u}_i\}$ 
27:    if  $\bar{u}_{i^*} < \bar{u}$  then
28:      Set  $\mathbb{S} = \mathbb{S} - \mathcal{S}_{i^*} \cup \mathcal{S}'_{i^*}$ 
29:      return True
30:    else
31:      return False
32:    end if
33:  end if
34: end function

```

Details of Algorithms 2. Algorithm 2 presents the main idea of balancing the cross-cluster upload and download

traffics. Let \mathcal{U} denote the residual chunks to be repaired and \mathcal{R} be the chunks selected to be repaired in a repair round.

Algorithm 2 finds the chunks to be repaired in a repair round via calling the REPAIR procedure (Lines 1-17). In each repair round, we first randomly select chunks from \mathcal{U} and construct an initial set of chunks (denoted by \mathcal{R}) to be repaired (Line 3). We use the symbol $|\mathcal{R}|$ to represent the number of chunks in \mathcal{R} . Therefore, $|\mathcal{R}|$ is always equal to r except the final repair round. For each chunk $H_i \in \mathcal{R}$, we then generate a valid repair solution \mathcal{S}_i to repair H_i (where $1 \leq i \leq |\mathcal{R}|$) and get a multi-stripe repair solution \mathbb{S} (Line 4). To balance the repair traffic, we give priority to calling the SUBSTITUTE function (Algorithm 3), which seeks to mitigate the traffic on the most loaded cluster through substituting a repair solution in \mathbb{S} with another valid one. If the SUBSTITUTE function cannot take effect, then we resort to the SWAP function (Algorithm 4), which tries to swap a chunk in \mathcal{R} with another chunk in \mathcal{U} to look for traffic reduction on the most loaded cluster (Lines 7-9). We repeat the balancing trial for t times (Lines 10-13), and finally obtain a set of chunks selected to be repaired in this repair round as well as their repair solutions. We then update \mathcal{U} by evicting the chunks that are selected in \mathcal{R} (Line 15).

In the MAIN procedure (Lines 18-25), we repeatedly call the REPAIR function until all the chunks in \mathcal{U} have been successfully scheduled for being repaired (Lines 19-23). Finally, we return the chunks to be repaired as well as the corresponding valid repair solutions in each repair round, where e is the number of repair rounds to be performed (Line 24).

Details of Algorithms 3. Algorithm 3 presents the detailed procedures of the SUBSTITUTE function. Given the multi-stripe repair solution \mathbb{S} , we can get the cross-cluster upload traffics loaded over the l clusters, denoted by $\{u_1, u_2, \dots, u_l\}$. Therefore, the most cross-cluster upload traffic among the l cluster is $\bar{u} = \max\{u_i | 1 \leq i \leq l\}$. Similarly, we can derive the most cross-cluster download traffic over the l clusters, denoted by \bar{d} (Line 2). If $\bar{d} > \bar{u}$, then the repair procedure is bottlenecked by the cross-cluster download traffic, which should be given priority in traffic balancing. We first pinpoint the cluster C_x that affords the most cross-cluster download traffic (Line 4), and locate every repair solution $\mathcal{S}_i \in \mathbb{S}$, satisfying that \mathcal{S}_i chooses C_x as the destination cluster. We then seek to find another valid repair solution \mathcal{S}'_i by substituting C_x in \mathcal{S}_i with another destination cluster (Line 7). By substituting \mathcal{S}_i with \mathcal{S}'_i , we can generate a new multi-stripe repair solution \mathbb{S}_i , and get its most cross-cluster download traffic (termed \bar{d}_i) (Line 8). Because \mathcal{S}'_i is also a valid repair solution, \mathbb{S}_i introduces the least cross-cluster repair traffic as well, implying that a smaller \bar{d}_i results in more balanced cross-cluster download traffic. We finally select the substitution that can produce the most balanced cross-cluster download traffic (Lines 11-14). If the substitution cannot further balance the cross-cluster download traffic, then the function returns false (Lines 15-16).

Balancing the cross-cluster upload traffic is similar (Lines 18-33), except that we will substitute the cluster that affords the most cross-cluster upload traffic with another cluster for reading data (Line 22). Finally, we will perform the substitution that introduces the most balanced cross-cluster upload traffic (Lines 26-29).

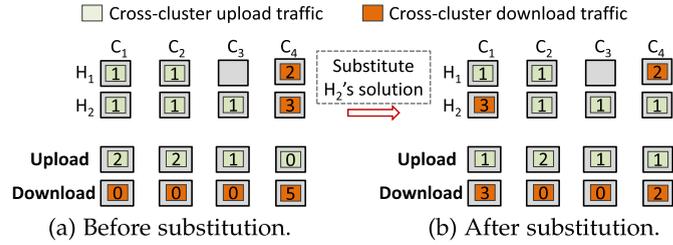


Fig. 5. Example of substitution. By substituting the repair solution of H_2 , we can reduce the most cross-cluster download traffic.

Example. Fig. 5 depicts a substitution example. Fig. 5a first shows the initial repair solutions of two chunks (namely H_1 and H_2). We can observe that C_1 and C_2 both afford the most cross-cluster upload traffic and need to send two chunks across clusters (i.e., $\bar{u} = 2$), while C_4 receives the most chunks across clusters (i.e., $\bar{d} = 5$). Therefore, the repair procedure is bottlenecked by the cross-cluster download traffic over C_4 . To balance the cross-cluster download traffic, we select C_1 to serve as the destination cluster of H_2 in Fig. 5b, and hence the most cross-cluster download traffic of the new multi-stripe repair solution is reduced to three chunks after substitution.

Algorithm 4. Swap Function

```

1: function Swap( $\mathcal{R}, \mathcal{S}, \mathcal{U}$ )
2:   Derive  $\bar{u}$  and  $\bar{d}$  from  $\mathcal{S}$ 
3:   // Find a chunk to be swapped
4:   if  $\bar{d} > \bar{u}$  then
5:     Get  $C_x$ , where  $d_x = \bar{d}$ 
6:     Find  $\mathcal{S}_i \in \mathcal{S}$  where  $\mathcal{S}_i$  selects  $C_x$  as the destination cluster
7:   else
8:     Get  $C_x$ , where  $u_x = \bar{u}$ 
9:     Find  $\mathcal{S}_i \in \mathcal{S}$  where  $\mathcal{S}_i$  reads data from  $C_x$  for repair
10:  end if
11:  // Find a chunk from  $\mathcal{U}$  and its repair solution
12:  for each chunk  $H_j \in \mathcal{U}$  do
13:    Find  $\mathcal{S}_j$  for  $H_j$ 
14:    Set  $\mathcal{S}_j = \mathcal{S} - \mathcal{S}_i \cup \mathcal{S}_j$ 
15:    Get  $\bar{u}_j$  and  $\bar{d}_j$  from  $\mathcal{S}_j$ 
16:    if  $\bar{u}_j \geq \bar{u}$  then
17:      continue
18:    end if
19:  end for
20:  Set  $j^* = \operatorname{argmin}_j \{\bar{d}_j\}$ 
21:  Set  $\mathcal{R} = \mathcal{R} - H_i \cup H_{j^*}$ 
22:  Set  $\mathcal{U} = \mathcal{U} - H_{j^*} \cup H_i$ 
23:  Set  $\mathcal{S} = \mathcal{S} - \mathcal{S}_i \cup \mathcal{S}_{j^*}$ 
24: end function

```

Details of Algorithms 4. Algorithm 4 further elaborates the procedures of the SWAP function. Given a multi-stripe repair solution \mathcal{S} , we use \bar{u} and \bar{d} to denote the most cross-cluster upload and download traffics over the l clusters, respectively (Line 2). If the repair is bottlenecked by the cross-cluster download traffic, then we can pinpoint the cluster C_x , which affords the most cross-cluster download traffic. We randomly choose a chunk $H_i \in \mathcal{R}$ for being swapped, satisfying that the repair solution of H_i chooses C_x as the destination cluster (Lines 4-6). Similarly, if the repair is bottlenecked by the cross-cluster upload traffic, then we

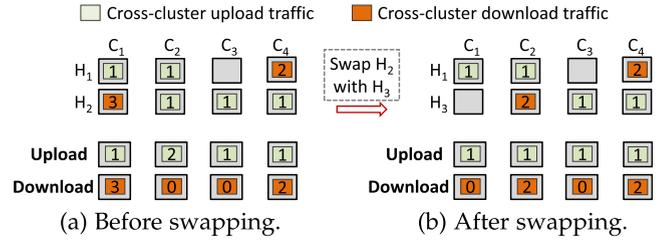


Fig. 6. Example of swapping. By swapping H_2 with H_3 , we can further balance the cross-cluster download traffic in this repair round.

turn to select the chunk that reads data from C_x for repair (Lines 8-9). We then consider each chunk $H_j \in \mathcal{U}$ for being swapped and measure the resulting cross-cluster upload and download traffics when temporarily swapping $H_i \in \mathcal{R}$ with H_j . We select the chunk $H_{j^*} \in \mathcal{U}$, such that swapping $H_i \in \mathcal{R}$ with H_{j^*} can reach the most balanced cross-cluster download traffic among all possible trials, while also producing more balanced cross-cluster upload traffic (Lines 12-20). We swap $H_i \in \mathcal{R}$ and $H_{j^*} \in \mathcal{U}$ and update the corresponding multi-stripe repair solution \mathcal{S} (Lines 21-23).

Example. Fig. 6 gives a swap example based on Fig. 5. To further balance the cross-cluster download traffic, we swap H_2 with another chunk $H_3 \in \mathcal{U}$, such that the most cross-cluster download traffic among the four clusters reduces from three chunks (Fig. 6a) to two chunks (Fig. 6b). The chunk H_2 will be re-organized in \mathcal{U} and scheduled in next repair rounds.

4.3 Complexity Analysis and Discussion

Complexity Analysis. Suppose that l is the number of clusters and r is the number of chunks repaired in a repair round. The complexity of Algorithm 1 is $O(l \log l)$. The complexity of Algorithm 3 is $O(rl)$. Let f be the total number of chunks to be repaired, then the complexity of Algorithm 4 is $O(f l \log l)$. Algorithm 2 calls the SUBSTITUTE and SWAP functions for at most t times in each repair round, therefore the complexity of Algorithm 2 is $O(et f l \log l)$, where e is the number of repair rounds.

Multi-Failure Repair. While ClusterSR mainly focuses on single node failure repair at present, it can be effortlessly extended to tackle multiple node failures. For example, we can simply repair each failed node individually by running ClusterSR directly. Besides, we can also schedule the repair of multiple nodes using ClusterSR through the following steps: 1) we first find the repair solutions that minimize the cross-cluster repair traffic for each failed chunk (i.e., based on Algorithm 1); 2) we then schedule the chunks to be repaired with the objective of balancing the cross-cluster upload and download repair traffic (through Algorithms 2, 3, and 4).

Degraded Read. ClusterSR can to some extent favor degraded read (i.e., requesting a chunk that fails coincidentally). When requesting a failed chunk in degraded read, ClusterSR can also minimize the amount of cross-cluster repair traffic by accessing the fewest clusters to collect k available chunks for serving the request (Lines 1-3 in Algorithm 1) and aggregate the selected chunks within the same cluster before cross-cluster transmission (Lines 14-15 in Algorithm 1).

Scenario With Limited Intra-Cluster Bandwidth. ClusterSR mainly focuses on the single node repair by minimizing and balancing the cross-cluster repair traffic, and therefore it is

more suitable to be deployed in the scenario with scarce cross-cluster bandwidth. For the scenario with limited intra-cluster bandwidth, we think that ClusterSR is also advantageous as it does not increase the intra-cluster bandwidth consumed while minimizing the cross-cluster repair traffic.

4.4 Reliability Analysis

We now analyze the reliability improvement achieved by ClusterSR. Data loss probability [33] and MTDL [16] are two different yet important metrics to evaluate the system reliability. In this paper, as the repair approaches differ in the time of repairing the failed data (as they will select different repair methods driven by their objectives), our major objective is to show that ClusterSR can vastly shorten the repair process and hence reduce the data loss probability during the single failure repair. We think that the data loss probability is straightforward and easily understood to reflect the influence of ClusterSR on the system reliability. We can also deduce that ClusterSR achieves longer MTDL as it has a larger repair rate (i.e., a shorter repair procedure).

Setting. Here, to simplify our analysis, we use rack to denote the cluster. We consider both node failures and cluster failures that might occur during the repair. Let θ_1 and θ_2 be the expected lifetimes of a node and a cluster, respectively. Suppose that the node and cluster failures are independent and their lifetimes are exponentially distributed. Such assumptions allow us to make simple and useful approximations [21]. The probability that a node fails (denoted by f_1) and the probability that a cluster fails (denoted by f_2) for a duration of time τ can be computed by

$$f_1 = 1 - e^{-\frac{\tau}{\theta_1}}, \quad f_2 = 1 - e^{-\frac{\tau}{\theta_2}}. \quad (3)$$

We then establish the values of the parameters of θ_1 and θ_2 based on field studies. For node failures, we set $\theta_1 = 10$ years [8]. For cluster failures, we mainly consider top-of-rack (ToR) switch failures. We can identify the average probability of a ToR switch failure in one year as 0.0278 from the field study [12, Fig. 4]. Therefore, based on Equation (3), we can estimate that $\theta_2 = 36$ years (by setting $f_2 = 0.0278$ and $\tau = 1$ year). After establishing the values of θ_1 and θ_2 , we calculate the approximate probabilities of node failure and cluster failure under different lengths of the observed period (i.e., τ).

Assumptions. To simplify our reliability analysis, we make the following assumptions. We focus on a storage system that organizes y nodes into l cluster, where each cluster consists of the same number of nodes (i.e., $\frac{y}{l}$, assuming that $\frac{y}{l}$ is an integer). Suppose that the data and parity chunk encoded by RS(n, k) are randomly distributed across the y nodes, while still ensuring cluster-level fault tolerance (Section 2.2). Consequently, the chunk distribution and the system configurations collectively determine the overall reliability of the stored data.

As chunks are randomly distributed across the whole clustered storage, we assume that the event with any than $n - k$ node failures will result in data loss for simplicity.

Comparison. In this analysis, we mainly compare ClusterSR with another two repair approaches, namely CAR [36] and RR (i.e., random repair). Specifically, CAR [36] aims to minimize and balance the cross-cluster upload

repair traffic in the dedicated repair, while RR randomly selects k surviving chunks for repair without minimizing and balancing the cross-cluster repair traffic. The detailed description of these two repair approaches can be referred to the second paragraph of Section 6.1.

Failure Events and Probabilities. We first consider a general number of node failures that might happen during the single failure repair. We use E_i to denote the event when any i remaining nodes fail during the single failure repair (where $0 \leq i \leq y - 1$), while there is no cluster failure. Therefore, we can compute the probability of E_i (denoted by $\Pr(E_i)$) as

$$\Pr(E_i) = \underbrace{\binom{y-1}{i} \cdot f_1^i \cdot (1-f_1)^{y-1-i}}_{i \text{ node failures}} \cdot \underbrace{(1-f_2)^l}_{\text{no cluster failure}}. \quad (4)$$

We next consider a general number of cluster failures that occur during the single failure repair. We use F_j to denote the event when j clusters fail (where $0 \leq j \leq l$), while the nodes in the $l - j$ clusters are still healthy. Therefore, the probability of F_j (denoted by $\Pr(F_j)$) can be given by

$$\Pr(F_j) = \underbrace{\binom{l}{j} \cdot f_2^j \cdot (1-f_2)^{l-j}}_{j \text{ cluster failures}} \cdot \underbrace{(1-f_1)^{(l-j)y/l}}_{\text{remaining nodes are healthy}}. \quad (5)$$

From Equations (3)~(5), we can observe that $\Pr(E_i)$ and $\Pr(F_j)$ closely relate to the repair time (i.e., τ), where the longer repair time (i.e., the larger τ) will result in the larger data loss probability.

Reliability Analysis for RS(9,6). We first consider the reliability for RS(9,6), as it is often used in production (e.g., QFS [28]). We consider the repair of 100 stripes encoded by RS(9,6) and assume that the chunks of these stripes are randomly dispersed across the clustered storage with $y = 16$ nodes and $l = 4$ clusters, such that each cluster is composed of $\frac{y}{l} = 4$ nodes. In the chunk placement, we also ensure that any cluster does not store more than $n - k$ chunks of any stripe, such that this configuration can tolerate any three-node failure or any single cluster failure.

Suppose that a node fails at this time. During the single failure repair, the clustered storage can still ensure data reliability in the face of the following failures: (i) no more than two node fail while there is no cluster failure (i.e., $\cup_{i=0}^2 E_i$), and (ii) only the cluster where the failed node resides fails, while all the nodes in the surviving clusters are still available. Therefore, the data loss probability can be deduced as

$$\Pr_{\text{dl}} = 1 - \left[\sum_{0 \leq i \leq 2} \Pr(E_i) + \frac{\Pr(F_1)}{l} \right].$$

Reliability Analysis for RS(16,12). We also consider the reliability of RS(16,12), since it is also considered in production systems (e.g., Windows Azure Storage [17]). Similar with the analysis of RS(9,6), we consider the repair of 100 stripes encoded by RS(16,12) in the clustered storage with $y = 20$ nodes and $l = 5$ clusters, where each cluster consists of $\frac{y}{l} = 4$ nodes. The placement can ensure data reliability under any four-node failure or any single cluster failure.

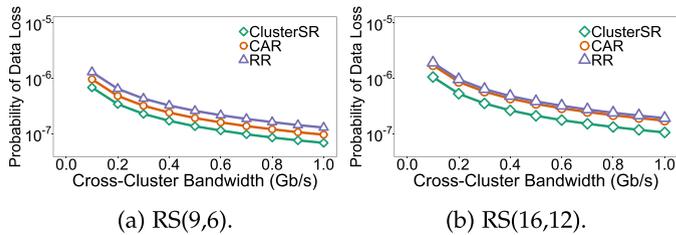


Fig. 7. Reliability analysis: Data loss probability.

Suppose a node fails at this time. The reliability analysis is similar with that of RS(9,6), except the difference that RS(16,12) can tolerate no more than three node failures during the single failure repair (i.e., $\cup_{i=0}^3 E_i$). Therefore, the data loss probability can be given by:

$$\Pr_{\text{dl}} = 1 - \left[\sum_{0 \leq i \leq 3} \Pr(E_i) + \frac{\Pr(F_1)}{l} \right].$$

Results. We finally investigate \Pr_{dl} under different repair approaches. Suppose that chunks of each stripe are randomly distributed across the whole cluster without violating the single cluster fault tolerance. Assuming that a node fails at this time, we then execute the three repair approaches, and measure the maximum cross-cluster upload and download traffics loaded among the clusters for each repair approach. Suppose that the repair procedure is seriously bottlenecked by the cross-cluster bandwidth. Therefore, the repair time can be simply deduced through dividing the maximum cross-cluster upload (resp. download) traffic by the cross-cluster bandwidth, once the cross-cluster upload (resp. download) traffic is the bottleneck of the repair process. We then set the chunk size as 64 MB and vary the cross-cluster bandwidth from 0.1 to 1 Gb/s. Finally, given the repair time (i.e., τ in the reliability analysis, see Equation (3)) of each repair approach, we enumerate every possible node and cluster failure event and calculate the data loss probability \Pr_{dl} . Fig. 7 presents the results.

We can observe that by minimizing and balancing the cross-cluster repair traffic, ClusterSR incurs the shortest repair process and thereby the smallest data loss probability among the three repair approaches. In addition, as CAR can reduce the cross-cluster repair traffic compared to RR, it induces less repair time and thereby a smaller data loss probability. Specifically, for RS(9,6), the average data loss probabilities of ClusterSR, CAR, and RR are 2.01×10^{-7} , 2.83×10^{-7} , and 3.81×10^{-7} , respectively.

5 IMPLEMENTATION

We have implemented a ClusterSR prototype in C++ with around 2,700 lines of code. We use Jerasure v.1.2 [30] to realize the encoding and decoding functionalities.

System Architecture. Fig. 8 presents the system architecture of the ClusterSR prototype. In particular, the ClusterSR prototype comprises a global *coordinator*, a *proxy* for every cluster, and an *agent* per storage node. The coordinator keeps track of the metadata information for each chunk, including the storage node that each chunk resides and the stripe identity that each chunk is organized into. When detecting a node failure, the coordinator first identifies the stripes that

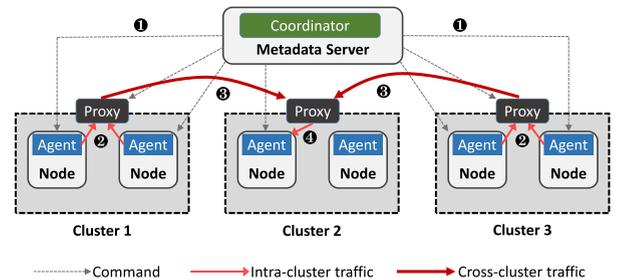


Fig. 8. System architecture of ClusterSR.

have the failed chunks and constructs repair solutions. It then issues the commands to the proxies and agents for instructing the repair procedure (step ① in Fig. 8). Upon receiving the commands, the agent will read the requested chunk from local storage and send it to the corresponding proxy within the same cluster (step ②). For each stripe, the proxy will aggregate the chunks received from the storage nodes within the same cluster, and send the resulting chunk to the proxy of the cluster where the destination node resides for aggregating the chunks received across clusters (step ③). Finally, the proxy sends the resulting chunk to the destination node for performing the repair operation (step ④). After all the chunks have been successfully repaired, the agents return acknowledgements to the coordinator.

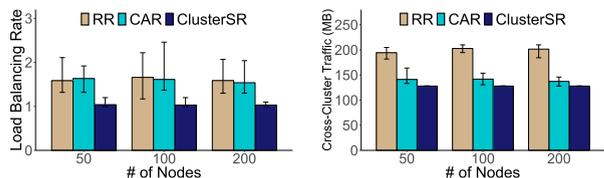
Multi-Threading. To improve the repair efficacy, we partition a chunk into many small fixed-size *packets* and use multi-threading to realize the *repair pipelining* as follows. For the agent that is to send data for repair, we create two threads, with one thread continually reads packets from the local storage and the other sends the packets. The proxy also creates multiple threads to receive packets, aggregate them, and send the resulting packets to the agent for final repair. If an agent is responsible for repairing the failed chunk, it will generate multiple threads to receive packets from the proxies, perform repair, and write each repaired packet to the local storage.

Integration With HDFS. Our ClusterSR prototype can be effortlessly integrated into state-of-the-art distributed storage systems. Here, we show how ClusterSR can assist the data repair in HDFS.¹ Specially, HDFS comprises a NameNode (for metadata management) and multiple DataNodes (for data storage). Therefore, we can deploy the coordinator in the NameNode, and run the agents in the DataNodes. Besides, we deploy a proxy in a DataNode for each cluster. The coordinator executes the command “`hdfs fsck / -files -blocks -locations`” in the NameNode to learn the location and stripe identity of each chunk. It then establishes the repair solutions for the failed chunks and guides the repair procedure by sending the commands to the involved proxies and agents. The integration needs *nomodification* to the HDFS codebase.

6 PERFORMANCE EVALUATION

We carry out extensive performance evaluation, in terms of large-scale simulation and testbed experiments on Alibaba Cloud ECS. The large-scale simulation and the testbed

1. In HDFS each data chunk is stored along with its metadata chunk. In this integration, we mainly focus on the repair of the data chunk.



(a) Load balancing rate. (b) Cross-cluster repair traffic.

Fig. 9. Experiment A.1 (Impact of number of nodes).

experiments are actually complementary to each other. In the large-scale simulation, we aim to learn the effectiveness of ClusterSR on reducing and balancing the cross-cluster repair traffic when ClusterSR is deployed in large-scale clusters (with hundreds of nodes). We then conduct testbed experiments to measure the real repair performance (denoted by the repair throughput) when ClusterSR is deployed in a real-world cloud data center. To summarize, the large-scale simulation and testbed experiments can well evaluate the overall performance of ClusterSR from different perspectives (i.e., the cross-cluster repair traffic, the load balancing rate, and the repair throughput).

We expect to answer the following questions.

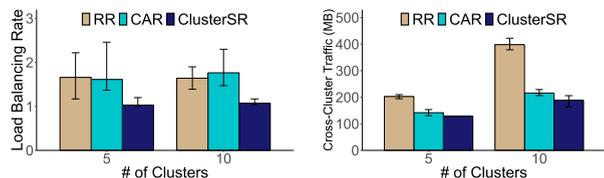
- Is ClusterSR effective on balancing the cross-cluster upload and download repair traffics? (Section 6.1)
- How sensitive ClusterSR is when any configuration varies? (Sections 6.1~6.3)
- How much cross-cluster repair traffic can be reduced by ClusterSR? (Section 6.1)
- How much computation time ClusterSR needs to establish the repair solution? (Section 6.3)

6.1 Large-Scale Simulation

We conduct simulations to unveil the performance of ClusterSR in large-scale storage clusters. We remove the storage and network I/O operations in our prototype, and evaluate the load balancing rate and cross-cluster repair traffic.

We compare ClusterSR with another two repair approaches, namely random repair (RR) and cross-rack-aware repair (CAR) [36]. RR randomly retrieves k out of the $(n - 1)$ surviving chunks for repair without concerning the clusters they reside. Therefore, RR can be treated as a baseline repair approach as it considers neither the reduction nor the balancing of the cross-cluster repair traffic. For fair comparison, we also allow RR to aggregate the requested chunks within the same cluster before cross-cluster transmission. CAR is originally designed to minimize the cross-cluster repair traffic in dedicated repair (i.e., storing all the repaired chunks in a dedicated node), but it only balances cross-cluster upload traffic. To compare ClusterSR with CAR fairly, we extend CAR to scattered repair by randomly choosing a node to store the repaired chunk while preserving cluster-level fault tolerance.

We adopt the following default configurations. We set the chunk size as 64 MB and generate 10,000 stripes that are encoded via RS(9,6) (also deployed in Quantcast File System [28]). These encoded stripes are then dispersed across five clusters with 100 nodes (i.e., 20 nodes per cluster), while promising the cluster-level fault tolerance. We repair $5 \times l$ chunks in each repair round, where l is the number of



(a) Load balancing rate. (b) Cross-cluster repair traffic.

Fig. 10. Experiment A.2 (Impact of number of clusters).

clusters. For both CAR and ClusterSR, we set the iteration steps as 50 to balance the cross-cluster repair traffic. We repeat each test for 10 runs and show the average values, as well as the maximum and minimum values in the figures (some may be invisible as the values are small).

Experiment A.1 (Impact of Number of Nodes). We first investigate the impact of the number of nodes. We vary the number of nodes from 50 to 200, and measure the load balancing rate and the induced cross-cluster repair traffic when repairing the chunks of a randomly selected node. Fig. 9 shows the results.

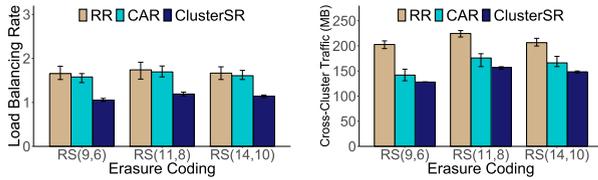
We can make two observations. First, ClusterSR has significant effect on balancing the cross-cluster upload and download traffics. In particular, the average load balancing rate of ClusterSR across all the test is 1.04, which closely approaches to the optimum value (i.e., 1). As a comparison, the average load balancing rates of RR and CAR are 1.61 and 1.59, respectively. We can observe that because of the negligence on the cross-cluster download traffic, even though CAR can well balance the cross-cluster upload traffic, it still has almost the same load balancing rate as RR, a repair approach that does not perform any load balancing operation at all. Besides, the load balancing rate of ClusterSR is much more stable than the other two approaches, implying that ClusterSR can still work well under different chunk distributions and system architectures.

Second, ClusterSR can reduce 8.6 and 36.8 percent of cross-cluster repair traffic compare to CAR and RR, respectively. This is because ClusterSR can minimize the cross-cluster repair traffic in scattered repair by carefully choosing chunks to be retrieved for repair and the destination node to store the repaired chunk (Section 4.1). Although CAR can minimize the cross-cluster repair traffic in dedicated repair [36], it cannot sustain its effectiveness in scattered repair.

Experiment A.2 (Impact of Number of Clusters). We then study how the number of clusters impact the load balancing rate and the cross-cluster repair traffic incurred. We perform the simulation when the default 100 nodes are organized into five clusters (i.e., with 20 nodes per cluster) and ten clusters (i.e., with 10 nodes per cluster), respectively. Fig. 10 depicts the results.

We can make two observations. First, ClusterSR retains its advantage on balancing the cross-cluster upload and download repair traffic under different numbers of clusters (Fig. 10a). The average load balancing rates of ClusterSR, CAR, and RR are 1.05, 1.65, and 1.69, respectively.

Second, the cross-cluster repair traffic significantly increases with the number of clusters (Fig. 10b). This is because when a clustered storage system has more clusters, each cluster will store fewer chunks of a stripe, such that the system has to access more clusters to collect enough surviving chunks for repair. Overall, ClusterSR reduces 9.7-12.8 and 36.9-52.7 percent of cross-cluster repair traffic compared to CAR and RR, respectively.



(a) Load balancing rate. (b) Cross-cluster repair traffic.

Fig. 11. Experiment A.3 (Impact of erasure coding).

Experiment A.3 (Impact of Erasure Coding). We also measure how different erasure codes influence the repair. We measure the load balancing rates and the cross-cluster repair traffics when the deployed erasure codes are set as RS(9,6), RS(11,8), and RS(14,10), respectively. Fig. 11 presents the results.

We can observe that ClusterSR can well balance the cross-cluster upload and download traffic under different erasure codes. The average load balancing rate of ClusterSR is 1.11.

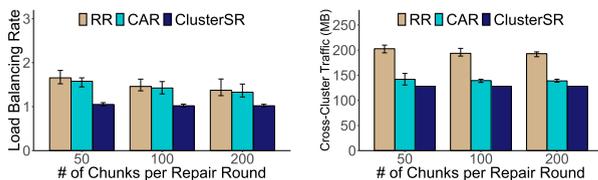
In addition, the amount of cross-cluster repair traffic is easily susceptible to the values of k and n in erasure coding. For example, ClusterSR incurs additional 23.6 percent of cross-cluster repair traffic when the system transitions from RS(9,6) to RS(11,8). The underlying reason is that compared to RS(9,6), a stripe of RS(11,8) will be stored in more clusters and the repair of RS(11,8) requires to take back more surviving chunks (i.e., $k = 8$).

Experiment A.4 (Impact of Number of Chunks Repaired per Round). We further study the impact of the number of chunks repaired in each repair round. We vary the number of chunks repaired in a round from 50 to 200, and measure the resulting load balancing rates and the cross-cluster repair traffics. Fig. 12 illustrates the results.

We can observe that the load balancing rate of ClusterSR stays nearly unchanged, whereas those of CAR and RR decline when more chunks are repaired in a round (Fig. 12a). However, repairing a large number of chunks simultaneously is usually not preferable, as it easily blows up the network and storage I/Os to the clustered storage system. Besides, the cross-cluster repair traffics of the three repair approaches are nearly constant when repairing different numbers of chunks in a round. This is reasonable as each chunk on the failed node is recovered independently.

Experiment A.5 (Impact of Iterative Steps). We then investigate the number of iterative trials (i.e., t in Algorithm 2) performed to substitute and swap repair solutions for opportunistically balancing the cross-cluster repair traffic. We vary the number of iterative steps from 5 to 100, and show the resulting load balancing rates in Table 1.

The load balancing rate of ClusterSR first sharply declines and then exhibits stable. The reason is that given a multi-stripe



(a) Load balancing rate. (b) Cross-cluster repair traffic.

Fig. 12. Experiment A.4 (Impact of number of chunks repaired in each round).

TABLE 1
Experiment A.5 (Impact of Number of Steps)

Steps	5	10	20	50	100
Load balancing rate	1.42	1.16	1.04	1.04	1.04

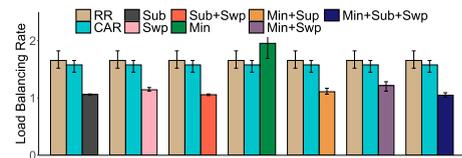
unbalanced repair solution, it is much easier to find a much more balanced one at the beginning. The optimization room becomes smaller with the increase of the iteration steps.

Experiment A.6 (Breakdown Analysis). We conduct the breakdown of ClusterSR and investigate the effectiveness of each technique in ClusterSR. Actually, the three techniques in ClusterSR can be abbreviated as follows: (i) the Min technique, which stands for the technique for minimizing the cross-cluster repair traffic (i.e., Algorithm 1); (ii) the Sub technique, which substitutes a single-stripe repair solution with another one for repairing the same chunk, so as to seek for a lower load balancing rate (i.e., Algorithm 3); and (iii) the Swp technique, which swaps a chunk selected to be repaired with another chunk that is not chosen yet to look for a lower load balancing rate (i.e., Algorithm 4). We try all the six possible combinations of the three approaches, and compare them to CAR and RR on the load balancing rate and the cross-cluster repair traffic. Fig. 13 shows the result, where ClusterSR is the synthesis of Min + Sub + Swp.

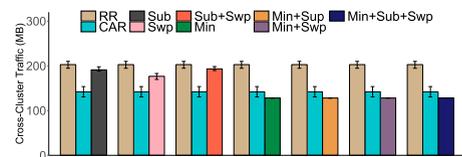
We can observe that Min + Sub + Swp (i.e., ClusterSR) is the sole one approach that achieves both of the lowest load balancing rate (see Fig. 13a) and the least cross-cluster repair traffic (see Fig. 13b) among all the approaches. For example, though the Min approach can minimize the cross-cluster repair traffic (Fig. 13b), it introduces the largest load balancing rate (Fig. 13a). As a comparison, the Sub and Swp approaches can well balance the cross-cluster repair traffic (Fig. 13a), but they induce a considerable amount of the cross-cluster repair traffic instead (Fig. 13b).

This experiment also indicates that the three techniques designed in ClusterSR are complementary mutually and do not compromise the effectiveness of each other.

Experiment A.7 (Effect on LRCs). We finally assess the performance of ClusterSR for non-RS codes. We mainly use LRC [17], [32] as an instance. Generally, LRC is configured by three parameters namely k , l , and g . LRC(k, l, g) takes k data chunks as input and generates another g global parity chunks; besides, it also breaks the k data chunks into l local groups

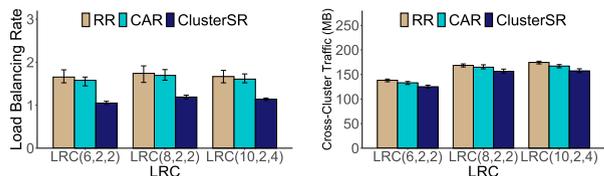


(a) Load balancing rate.



(b) Cross-cluster repair traffic.

Fig. 13. Experiment A.6 (Breakdown analysis).



(a) Load balancing rate. (b) Cross-cluster repair traffic.

Fig. 14. Experiment A.7 (Effect on LRCs).

with k/l data chunks per local group (assume that k is divisible by l) and keeps a local parity chunk for each group. We choose LRC(6,2,2), LRC(8,2,2), and LRC(10,2,4) in this evaluation, which are also considered in previous work [20]. For each stripe encoded by LRC, we ensure that each cluster stores at most $g + 1$ chunks to promise the cluster-level fault tolerance. The remaining configurations (including the number of nodes and the number of chunks repaired in each round) are consistent with the tests in evaluating the RS codes (see the third paragraph of Section 6.1). Fig. 14 depicts the load balancing rates and cross-cluster repair traffics.

ClusterSR remains effective on minimizing and balancing the cross-cluster repair traffic for different LRCs. Specifically, the average load balancing rate of the three LRCs achieved by ClusterSR is 1.15; besides, ClusterSR can reduce the cross-cluster repair traffic by 8.8 and 5.6 percent compared to RR and CAR, respectively.

6.2 Testbed Experiments

We further evaluate ClusterSR on Alibaba Cloud ECS [1] to study its performance in real-world cloud environment. We set up 21 virtual machine instances of type `ecs.g6.large` in East China region (Hangzhou Zone H). Each instance is equipped with 2 vCPUs (2.5 GHz Intel Xeon Platinum), 8 GB memory, and 40 GB ultra-disk space. The operating system is Ubuntu 14.04 and the network bandwidth that each instance can achieve is around 3 Gb/s (measured by `iperf`).

To run ClusterSR on the 21 instances, we deploy the coordinator on one instance and organize the remaining 20 instances into four clusters (i.e., five instances per cluster). For each cluster, we run the agents on four instances and deploy the proxy on the last instance. To mimic the network bandwidth diversity, we use the Linux traffic control tool `tc` to throttle the network bandwidth among proxies.

We use the following default configurations. We use RS(9,6) as the default erasure code, and set the chunk size and packet size as 64 and 4 MB, respectively. The cross-cluster bandwidth is set as 0.15 Gb/s. We then generate the stripes and randomly distribute their chunks across the cluster. We repair 100 chunks in total by performing five repair rounds, where each repair round repairs 20 chunks. We measure the overall duration starting from the time when the coordinator detects a node failure until the time when all the lost chunks are all repaired. We then calculate the *repair throughput* (i.e., the size of data that can be repaired per second) via dividing the size of the repaired data by the duration time. We repeat each test for five runs and plot the average result as well as the error bars showing the maximum and the minimum in the test.

Experiment B.1 (Impact of Cross-Cluster Bandwidth). We first measure the repair throughput when the cross-cluster bandwidth is varied as 0.1, 0.15, and 0.3 Gb/s. Fig. 15a

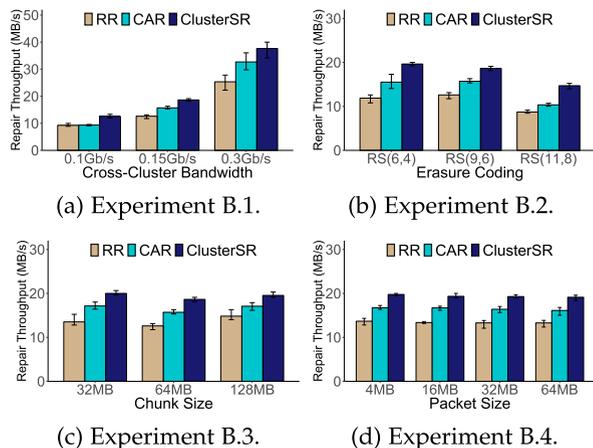


Fig. 15. Experiments on Alibaba Cloud ECS.

shows the results. We can derive the following findings. First, ClusterSR can improve the repair throughput by 15.2-34.3 and 35.4-48.6 percent when compared with CAR and RR, respectively. This is because ClusterSR can both minimize and balance the cross-cluster upload and download traffics for repair. Second, the repair throughput increases with the cross-cluster bandwidth, demonstrating that the repair process is seriously restricted by the scarce cross-cluster bandwidth. This observation reveals the necessity of minimizing the cross-cluster repair traffic.

Experiment B.2 (Impact of Different Erasure Codes). We next study how erasure coding affects the repair throughput. We select RS(6,4), RS(9,6), and RS(11,8), and measure the repair throughput for different erasure codes. Fig. 15b shows the results. The repair throughput of all the three approaches decreases when the value k becomes larger. For example, the repair throughput of ClusterSR decreases from 19.6 MB/s (when $k = 4$) to 14.7 MB/s (when $k = 8$). This is generally because when k increases, we have to retrieve more chunks for repair, thereby adding more computation and network transmission latencies. Second, ClusterSR can accelerate the repair process by 26.6-42.1 and 47.8-68.8 percent when compared with CAR and RR, respectively.

Experiment B.3 (Impact of Chunk Size). We further evaluate the repair throughput under different chunk sizes. Fig. 15c shows the results. The repair throughput is stable when the chunk size changes. Overall, ClusterSR improves the repair throughput by 14.4-18.7 and 31.8-47.9 percent when compared with CAR and RR, respectively.

Experiment B.4 (Impact of Packet Size). We vary the packet size from 4 to 64 MB (i.e., the chunk size) and measure the resulting repair throughputs of the three approaches. Fig. 15d shows the results. We can see that the repair throughputs are relatively stable under different packet sizes. For example, the repair throughput of ClusterSR is 19.8 MB/s when the packet size is 4 MB, which remains 19.1 MB/s when the packet size increases to 64 MB. We suspect the scarcity of the cross-cluster bandwidth is the underlying reason, which makes the time spent in cross-cluster data transfer take up the majority of the repair time. Consequently, the packet size has marginal impact on the resulting repair time. Overall, ClusterSR can accelerate the repair process by 17.5 and 44.5 percent compare to CAR and RR, respectively.

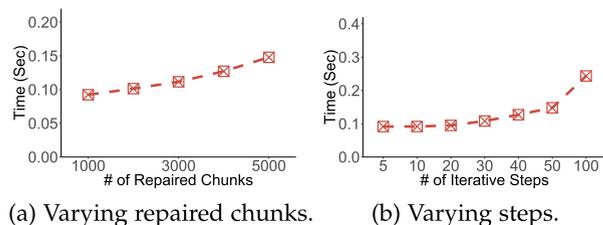


Fig. 16. Microbenchmarks.

6.3 Microbenchmarks

ClusterSR generates the multi-stripe repair solutions for instructing the surviving nodes for repair. Therefore, the time needed to establish the repair solutions is rather crucial to fulfill the online repair. We allocate one instance on Alibaba Cloud ECS [1] with the same configuration as those in the testbed experiments. We generate 200,000 stripes encoded by RS(9,6) and randomly distribute them in a 100-node data center with five clusters. We suppose to repair 50 chunks in a repair round. We then vary the total number of chunks to be repaired and the number of iterative steps, and measure the resulting time to generate the repair solutions.

Fig. 16a shows that ClusterSR is efficient to derive the repair solutions. For example, it merely needs about 0.15 seconds to obtain the repair solutions for repairing 5,000 chunks. As ClusterSR only incurs extremely lightweight computation overhead, it can be deployed in the online repair scenario.

Fig. 16b indicates that the time for ClusterSR to generate the repair solutions increases with the number of iterative steps performed (i.e., t in Algorithm 2). However, as the load balancing rate becomes stable when the number of iterative steps is around 30 (see Experiment A.5), ClusterSR merely needs about 0.11 seconds to generate the repair solutions with near-optimal load balancing rate.

7 RELATED WORK

Repair-Efficient Codes. Some repair-efficient codes are designed to suppress the repair traffic. LRCs [17], [32] associate a subset of data chunks of a stripe with a local parity chunk, thereby trading additional storage for reduced repair traffic. Regenerating codes [9], [29] employ the subpacketization technique and allow surviving nodes to send the subchunks calculated from the locally stored data. As an independent study, ClusterSR can work for different erasure codes to achieve fast repair in clustered storage.

Repair Scheduling. Some studies schedule the repair by fully utilizing the available bandwidth. PPR [24] decomposes a repair operation into partial repair sub-operations that are parallelized across multiple nodes. CAR [36] minimizes the cross-cluster repair traffic in data centers by accessing the minimum number of clusters in each chunk's repair. ECPipe [22] partitions a chunk into small-size slices and pipelines the repair of slices across nodes to achieve $O(1)$ repair time. DoubleR [16] performs both intra-cluster and cross-cluster regenerations to minimize the cross-cluster repair traffic in hierarchical data centers. These studies focus on dedicated repair, while ClusterSR balances the cross-cluster upload and download traffics in scattered repair.

Parity Declustering. Parity declustering [14], [25] distributes stripes across different nodes, with the aim of exploiting

the available resources of the entire system in repair. One similar approach is applied in RAMCloud [27], a replication-based storage system that scatters the replicas across the system for fast repair. FastPR [34] couples migration and repair to fully leverage the I/O of the soon-to-fail node and parallelize the repair across the whole storage cluster. As a comparison, ClusterSR focuses on the scattered repair in the cluster storage with bandwidth diversity phenomenon.

8 CONCLUSION

We consider the scattered repair in clustered storage and propose ClusterSR, a cluster-aware scattered repair approach. By carefully examining the data distribution, ClusterSR first finds the valid repair solutions that achieve the least cross-cluster repair traffic for each failed chunk. ClusterSR then constructs multi-stripe repair solutions to balance the cross-cluster upload and download traffics for repair. We evaluate ClusterSR via both large-scale simulation and Alibaba Cloud ECS experiments, and demonstrate that ClusterSR can well suppress and balance the induced cross-cluster repair traffic, and hence improve the repair throughput.

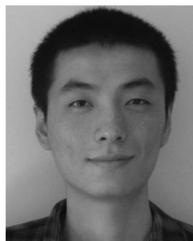
ACKNOWLEDGMENTS

This work was supported in part by NSFC under Grant 62072381, Grant 61832011, Grant 61702569, and Grant 61702013, in part by CCF-Tencent Open Fund WeBank Special Fund, and in part by Fujian Provincial Natural Science Foundation under Grant 2020J01002. A preliminary version [35] of this article was presented at the 34th IEEE International Parallel and Distributed Processing Symposium (IPDPS'20).

REFERENCES

- [1] Alibaba Cloud Elastic Compute Service, 2009. [Online]. Available: <https://www.alibabacloud.com/product/ecs>
- [2] Facebook HDFS, 2014. [Online]. Available: <https://github.com/facebookarchive/hadoop-20>
- [3] Apache Hadoop 3.1.1, 2018. [Online]. Available: <https://hadoop.apache.org/docs/r3.1.1/>
- [4] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar, "ShuffleWatcher: Shuffle-aware scheduling in multi-tenant Map-Reduce clusters," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2014, pp. 1–12.
- [5] Y. Chen, *et al.*, "Giza: Erasure coding objects across global data centers," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 539–551.
- [6] M. Chowdhury, S. Kandula, and I. Stoica, "Leveraging endpoint flexibility in data-intensive clusters," in *Proc. ACM SIGCOMM Conf.*, 2013, pp. 231–242.
- [7] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica, "HUG: Multi-resource fairness for correlated and elastic demands," in *Proc. 13th USENIX Conf. Netw. Syst. Des. Implementation*, 2016, pp. 407–424.
- [8] A. Cidon, R. Escriva, S. Katti, M. Rosenblum, and E. Sirer, "Tiered replication: A cost-effective alternative to full cluster geo-replication," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2015, pp. 31–43.
- [9] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran, "Network coding for distributed storage systems," *IEEE Trans. Inf. Theory*, vol. 56, no. 9, pp. 4539–4551, Sep. 2010.
- [10] D. Ford *et al.*, "Availability in globally distributed storage systems," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 61–74.
- [11] S. Ghemawat, H. Gobioff, and S. Leung, "The Google file system," in *Proc. 9th ACM Symp. Operating Syst. Princ.*, 2003, pp. 29–43.
- [12] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. ACM SIGCOMM Conf.*, 2011, pp. 350–361.

- [13] A. Greenberg, et al., "VL2: A scalable and flexible data center network," in *Proc. ACM SIGCOMM Conf. Data Commun.*, 2009, pp. 51–62.
- [14] M. Holland, G. Gibson, and D. Siewiorek, "Architectures and algorithms for on-line failure recovery in redundant disk arrays," *Distrib. Parallel Databases*, vol. 2, no. 3, pp. 295–335, 1994.
- [15] H. Hou, P. P. C. Lee, K. W. Shum, and Y. Hu, "Rack-aware regenerating codes for data centers," *IEEE Trans. Inf. Theory*, vol. 65, no. 8, pp. 4730–4745, Aug. 2019.
- [16] Y. Hu et al., "Optimal repair layering for erasure-coded data centers: From theory to practice," *ACM Trans. Storage*, vol. 13, no. 4, 2017, Art. no. 33.
- [17] C. Huang et al., "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2012, Art. no. 2.
- [18] H. Jin et al., "Approximate code: A cost-effective erasure coding framework for tiered video storage in cloud systems," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, Art. no. 95.
- [19] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, Art. no. 20.
- [20] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg, "On fault tolerance, locality, and optimality in locally repairable codes," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2018, pp. 865–877.
- [21] G. Lefebvre and M. J. Feeley, "Separating durability and availability in self-managed storage," in *Proc. 11th Workshop ACM SIGOPS Eur. Workshop*, 2004, pp. 28–es.
- [22] R. Li, X. Li, P. Lee, and Q. Huang, "Repair pipelining for erasure-coded storage," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf.*, 2017, pp. 567–579.
- [23] H. Liu et al., "Scheduling techniques for hybrid circuit/packet networks," in *Proc. 11th ACM Conf. Emerg. Netw. Experiments Technol.*, 2015, Art. no. 41.
- [24] S. Mitra, R. Panta, M. Ra, and S. Bagchi, "Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage," in *Proc. 11th Eur. Conf. Comput. Syst.*, 2016, Art. no. 30.
- [25] R. Muntz and J. Lui, "Performance analysis of disk arrays under failure," in *Proc. 16th Int. Conf. Very Large Data Bases*, 1990, pp. 162–173.
- [26] S. Muralidhar et al., "f4: Facebook's warm BLOB storage system," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 383–398.
- [27] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMcloud," in *Proc. 23rd ACM Symp. Operating Syst. Princ.*, 2011, pp. 29–41.
- [28] M. Ovsianikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The quantcast file system," *Proc. VLDB Endowment*, vol. 6, no. 11, pp. 1092–1101, 2013.
- [29] L. Parnies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. Gad, and Z. Bandic, "Opening the chrysalis: On the real repair performance of MSR codes," in *Proc. 14th Usenix Conf. File Storage Technol.*, 2016, pp. 81–94.
- [30] J. Plank, S. Simmerman, and C. Schuman, "Jerasure: A library in C/C++ facilitating erasure coding for storage applications-version 1.2," Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08-627, 2008.
- [31] I. Reed and G. Solomon, "Polynomial codes over certain finite fields," *J. Soc. Ind. Appl. Math.*, vol. 8, no. 2, pp. 300–304, 1960.
- [32] M. Sathiamoorthy, et al., "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, no. 5, pp. 325–336, 2013.
- [33] Z. Shen and P. Lee, "Cross-rack-aware updates in erasure-coded data centers," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, Art. no. 80.
- [34] Z. Shen, X. Li, and P. P. C. Lee, "Fast predictive repair in erasure-coded storage," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2019, pp. 556–567.
- [35] Z. Shen, J. Shu, Z. Huang, and Y. Fu, "ClusterSR: Cluster-aware scattered repair in erasure-coded storage," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2020, pp. 42–51.
- [36] 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.*, 2016, pp. 323–334.
- [37] I. Tamo and A. Barg, "A family of optimal locally recoverable codes," *IEEE Trans. Inf. Theory*, vol. 60, no. 8, pp. 4661–4676, Aug. 2014.
- [38] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *Proc. Int. Workshop Peer-to-Peer Syst.*, 2002, pp. 328–337.
- [39] X. Xie, et al., "AZ-code: An efficient availability zone level erasure code to provide high fault tolerance in cloud storage systems," in *Proc. 35th Symp. Mass Storage Syst. Technol.*, 2019, pp. 230–243.



Zhirong Shen (Member, IEEE) received the BS degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2010, and the PhD degree in computer science from Tsinghua University, Beijing, China, in 2016. He is currently an associate professor with Xiamen University. His current research interests include storage reliability and storage security.



Shiyao Lin received the BS degree from Fuzhou University, Fuzhou, China, in 2019. Currently, she is working toward the MS degree in computer science at Xiamen University, Xiamen, China. Her research interests include storage reliability and security.



Jiwu Shu (Fellow, IEEE) received the PhD degree in computer science from Nanjing University, Nanjing, China, in 1998, and finished the postdoctoral position research with Tsinghua University, in 2000. Since then, he has been teaching with Tsinghua University. His current research interests include storage security and reliability, non-volatile memory based storage systems, and parallel and distributed computing.



Chengxin Xie is currently working toward the undergraduate degree at Xiamen University, Xiamen, China. His research interests include persistent memory and storage reliability technologies.



Zhijie Huang received the PhD degree in computer science and technology from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016. He is currently a postdoctoral research associate with the Computer Science and Engineering Department, University of Texas at Arlington (UTA). His research interests include coding theory and its application, dependable and secure storage systems, and embedded systems.



Yingxun Fu (Member, IEEE) received the BS degree from North China Electric Power University, Beijing, China, the master's degree from the Beijing University of Posts and Telecommunications, Beijing, China, and the PhD degree from Tsinghua University, Beijing, China, in 2015. He is currently a professor with the North China University of Technology. His current research interests include storage reliability, in-memory computing, and quantum computing.

▽ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.