ClusterSR: Cluster-Aware Scattered Repair in Erasure-Coded Storage

Zhirong Shen*, Jiwu Shu†, Zhijie Huang‡, Yingxun Fu§
*School of Informatics, Xiamen University
†Department of Computer Science and Technology, Tsinghua University
‡Department of Computer Science and Engineering, The University of Texas at Arlington
§School of Computer Science, North China University of Technology
shenzr@xmu.edu.cn, shujw@tsinghua.edu.cn, zhijie.huang@uta.edu, fuyx@ncut.edu.cn

***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 6.7-52.7% of the cross-cluster repair traffic and improve 14.1-68.8% of the repair throughput.**

I. 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 [17], [27]. To guarantee data reliability against failures, pre-storing additional data redundancy is a commonly adopted approach in production systems [2], [10], [22]–[24], where replication and erasure coding are two representatives. Compared to replication, erasure coding [2], [16], [22], [24] is much more storage-efficient, which can attain the same degree of fault tolerance with far less storage redundancy [34]. 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 data 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 [15], and Facebook F4 [22].

While being storage-efficient, erasure coding incurs substantial *repair traffic* (i.e., data retrieved for repair). For example, Reed-Solomon codes (RS codes) [28], 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 II-B). 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], [9], [14], [30]. 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 MapReduce jobs [4]), is often oversubscribed and is shown to be much more scarce than the intra-cluster bandwidth (Section II-A). 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 [13], [14] 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 [32]. 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 the cross-cluster repair traffic in scattered repair as well. On the other hand, as NICs (network interface cards) and network cables extensively support *full duplex* transmission [7], [19], 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.

We therefore present ClusterSR, a **Cluster**-aware **S**cattered **R**epair 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*

1530-2075/20/$31.00 ©2020 IEEE
DOI 10.1109/IPDPS47924.2020.00015

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 6.7-52.7% of the cross-cluster repair traffic and improve 14.1-68.8% of the repair throughput. We also demonstrate that ClusterSR is effective on balancing the cross-cluster upload and download traffics.

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

II. Background

A. 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 [30], [32] or even a data center [5]. Figure 1 depicts the architecture of the clustered storage. Such architecture has been applied in modern data center deployment [9], [22] and assumed in previous work [6], [14], [30], [32].

The hierarchical architecture results in the *bandwidth diversity* phenomenon, where the cross-cluster bandwidth is often oversubscribed [4], [6], [11] 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 [11].

B. Erasure Coding

Erasure coding often operates on *chunks*, which are a collection of fixed-size information in units of MBs (e.g., 64MB by default in Hadoop HDFS [3]). In this paper, we mainly focus on the linear codes, including RS codes [28], regenerating codes [8], [25], and locally repairable codes (LRCs) [15], [29], [33]. 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 IV-A).

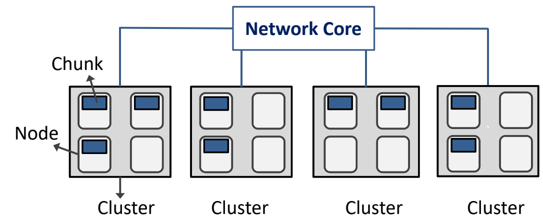


Fig. 1. Example of a clustered storage system deployed with RS$(9, 6)$.

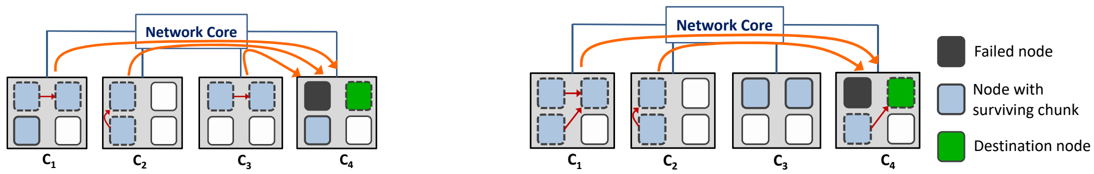
RS codes often use two parameters, namely k and n (where $k < n$), to configure their storage efficiency and fault-tolerance capability, which can be denoted by RS(n, k). In the encoding stage, RS(n, k) takes k data chunks as input and generates additional $n - k$ parity chunks via linear arithmetics over Galois finite field [28]. 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. Figure 1 shows that the nine chunks of a stripe encoded by RS$(9, 6)$ (i.e., $n = 9$ and $k = 6$) is 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 that any single cluster failure is tolerated.

C. Repair in Erasure Coding

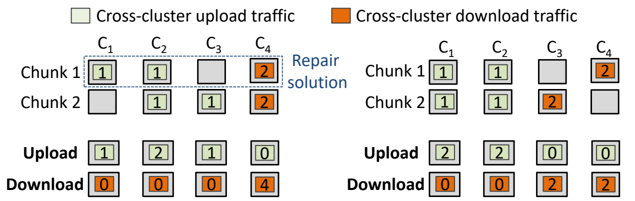
Repairing in erasure coding is an I/O intensive operation. For instance, RS(n, k) requires retrieving k surviving chunks to repair a chunk, indicating that the storage and network I/Os for repair are k times the size of the failed chunk. To improve the repair efficiency, regenerating codes [8], [25] 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 [25] 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 [15], [29], [33] 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



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



(a) Unbalanced repair traffic. (b) More balanced repair traffic.

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

same local group. Hence, LRCs can repair a chunk by merely retrieving the surviving chunks of the same local group.

III. Observations and Problem Formulation

A. Observations

Given the scarcity of the cross-bandwidth, we have the following two observations. To clarify, we use the clustered storage system in Figure 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 Figure 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 Figure 1 uses RS$(9, 6)$, it requires to retrieve any six surviving chunks to perform repair.

Figure 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 (Figure 2(a)) retrieves six surviving chunks from $\{C_1, C_2, C_3\}$ and stores the repaired chunk in C_4. By relying 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 [32]) and therefore only needs to send one chunk to C_4 [32]. Consequently, Figure 2(a) transmits three chunks across clusters for repair. As a comparison, the repair solution in Figure 2(b) 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, Figure 2(b) 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. Figure 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 Figure 3 is based on Figure 2(b), 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. Figure 3(a) and Figure 3(b) 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 Figure 3(a) and needs to download four chunks from other clusters. By contrast, each of the four clusters in Figure 3(b) 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 most cross-cluster upload or download traffic. Therefore, a repair solution with balanced cross-cluster upload and download traffics can well shorten the repair process.

B. Modeling

We further formulate the repair problem based on the following assumptions. First, the computation time in repair is often trivial [17], [32] and can be negligible. Second, as each node can be attached with multiple disks, the cumulative 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], [11], we assume that the cross-cluster transmission dominates the network transmission. Fourth, we focus on single failure, which accounts for more than 90% of all the failure events in practical storage deployment [17], [27].

Suppose that the clustered storage system consists of l clusters termed $\{C_1, C_2, \cdots, C_l\}$ 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 $\overline{u}$ and $\overline{d}$, where $\overline{u} = \max\{u_i | 1 \leq i \leq l\}$ and $\overline{d} = \max\{d_i | 1 \leq i \leq l\}$, respectively. Obviously, the repair time is determined by $m = \max\{\overline{u}, \overline{d}\}$ and can be given by $T = \frac{m}{b}$. As the cross-cluster upload and download traffics have equal size (i.e., $\sum_{i=1}^{l} u_i = \sum_{i=1}^{l} d_i$), then 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:

$$T = \frac{m}{b} \geq \frac{a}{b}. \tag{1}$$

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

$$T = \frac{m}{b} \geq \frac{a}{b} \geq \frac{a^*}{b} \tag{2}$$

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., $a = a^*$).

C. 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.

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

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.

IV. 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 IV-A). 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 IV-B).

A. 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 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 $\{C'_1, C'_2, \cdots, C'_l\}$, 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).

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, ..., C'_l}, where h_1 ≥ h_2 ≥ ... ≥ h_l
3: Establish a smallest value v, such that ∑_{1≤i≤v} h_i ≥ k
4: // Find a destination cluster
5: for 1 ≤ i ≤ 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 ≤ i ≤ v} ∪ C*
13: Select a destination node from C*
14: for 1 ≤ i ≤ 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
```

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 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 Figure 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 Figure 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

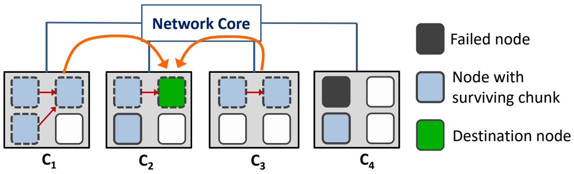


Fig. 4. A repair solution with the least repair traffic for RS$(9, 6)$.

(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 [32]. All valid repair solutions of a stripe will be considered when trying to balance the cross-cluster repair traffic (Section IV-B).

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 first show that the repair must access at least v clusters for reading sufficient data (as proved in CAR [32]). Therefore, the selection of C^* directly determines the cross-cluster repair traffic. There are two possibilities in selecting C^*. If each C'_i in $\{C'_1, C'_2, \cdots, C'_v\}$ has exactly $n-k$ surviving chunks of the failed stripe, then $C^* \notin \{C'_1, C'_2, \cdots, C'_v\}$ 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 breaking the cluster-level fault tolerance.

On the other hand, if some cluster C_i in $\{C'_1, C'_2, \cdots, C'_v\}$ stores less than $n-k$ surviving chunks of the failed stripe, then we have $C^* \in \{C'_1, C'_2, \cdots, C'_v\}$. In this case, each C_i (where $1 \leq i \neq d \leq v$) 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 is to say, 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.

Extension: Though Algorithm 1 mainly focuses on RS codes, it can be effortlessly tuned for LRCs and regenerating codes (Section II-C). Due to page limits, we only use LRCs as an instance. For LRCs, to repair a chunk by using the corresponding local parity chunk, we can sort the clusters based on the number of surviving chunks of the local group where the failed chunk resides, 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 [15] 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.

B. 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 Algorithm 2, Algorithm 3, and Algorithm 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. Algorithm 3 and Algorithm 4 elaborate the substitution and swapping procedures, respectively.

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(𝒰)
2:      // Initialize a set of chunks to be repaired
3:      Set ℛ ⊂ 𝒰
4:      Construct 𝕊 = {𝒮_1, 𝒮_2, ··· , 𝒮_|ℛ|} for ℛ
5:      // Balance the cross-cluster repair traffic
6:      while true do
7:          if SUBSTITUTE(ℛ, 𝕊) equals False then
8:              SWAP(ℛ, 𝕊, 𝒰)
9:          end if
10:         Set t = t − 1
11:         if t = 0 then
12:             break
13:         end if
14:     end while
15:     Set 𝒰 = 𝒰 − ℛ
16:     return (ℛ, 𝕊, 𝒰)
17: end function
18: procedure MAIN(𝒰)
19:     Initialize e = 0
20:     while 𝒰 ≠ φ do
21:         Set e = e + 1
22:         (ℛ_e, 𝕊_e, 𝒰) = REPAIR(𝒰)
23:     end while
24:     return {(ℛ_1, 𝕊_1), ··· , (ℛ_e, 𝕊_e)}
25: end procedure
```

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

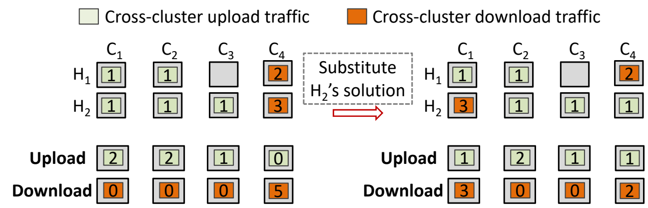
Algorithm 3 Substitute Function

1: **function** SUBSTITUTE($\mathcal{R}$, $\mathbb{S}$)
2: Derive $\overline{u}$ and $\overline{d}$ from $\mathbb{S}$
3: **if** $\overline{d} > \overline{u}$ **then**
4: Get C_x, where $d_x = \overline{d}$
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 $\overline{d}_i$ from $\mathbb{S}_i$
9: **end if**
10: **end for**
11: Set $i^* = arg\min_i\{\overline{d}_i\}$
12: **if** $\overline{d}_{i^*} < \overline{d}$ **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 $u_x = \overline{u}$
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 $\mathbb{S}_i = \mathbb{S}_i - \mathcal{S}_i \cup \mathcal{S}'_i$, and get $\overline{u}_i$ from $\mathbb{S}_i$
24: **end if**
25: **end for**
26: Set $i^* = arg\min_i\{\overline{u}_i\}$
27: **if** $\overline{u}_{i^*} < \overline{u}$ **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**

(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 $\{u_1, u_2, \cdots, u_l\}$. Therefore, the most cross-cluster upload traffic among the l cluster is $\overline{u} = \max\{u_i | 1 \leq i \leq l\}$. Similarly, we can derive



(a) Before substitution.

(b) After substitution.

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

the most cross-cluster download traffic over the l clusters, denoted by $\overline{d}$ (Line 2). If $\overline{d} > \overline{u}$, 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 $\overline{d}_i$) (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 $\overline{d}_i$ 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).

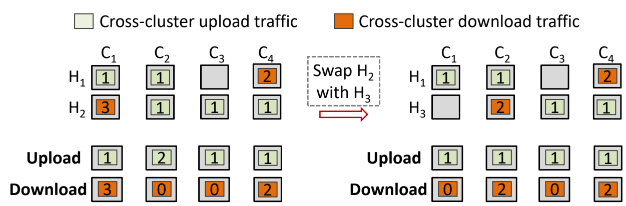
Balancing the cross-cluster upload traffic is similar (Lines 18-33), except that we will substitute the cluster that affords the most cross-cluster upload traffic with another cluster for reading data (Line 22). Finally, we will perform the substitution that introduces the most balanced cross-cluster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

Example: Figure 5 depicts a substitution example. Figure 5(a) 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., $\overline{u} = 2$), while C_4 receives the most chunks across clusters (i.e., $\overline{d} = 5$). 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 Figure 5(b), and hence the most cross-cluster download traffic of the new multi-stripe repair solution is reduced to three chunks after substitution.

Details of Algorithms 4: Algorithm 4 further elaborates the procedures of the SWAP function. Given a multi-stripe repair solution $\mathbb{S}$, we use $\overline{u}$ and $\overline{d}$ 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-

Algorithm 4 Swap Function

1: **function** SWAP($\mathcal{R}$, $\mathbb{S}$, $\mathcal{U}$)
2: Derive $\overline{u}$ and $\overline{d}$ from $\mathbb{S}$
3: `// Find a chunk to be swapped`
4: **if** $\overline{d} > \overline{u}$ **then**
5: Get C_x, where $d_x = \overline{d}$
6: Find $\mathcal{S}_i \in \mathbb{S}$ where $\mathcal{S}_i$ selects C_x as the destination cluster
7: **else**
8: Get C_x, where $u_x = \overline{u}$
9: Find $\mathcal{S}_i \in \mathbb{S}$ 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 $\mathbb{S}_j = \mathbb{S} - \mathcal{S}_i \cup \mathcal{S}_j$
15: Get $\overline{u}_j$ and $\overline{d}_j$ from $\mathbb{S}_j$
16: **if** $\overline{u}_j \geq \overline{u}$ **then**
17: **continue**
18: **end if**
19: **end for**
20: Set $j^* = arg\min_j\{\overline{d}_j\}$
21: Set $\mathcal{R} = \mathcal{R} - H_i \cup H_{j^*}$
22: Set $\mathcal{U} = \mathcal{U} - H_{j^*} \cup H_i$
23: Set $\mathbb{S} = \mathbb{S} - \mathcal{S}_i \cup \mathcal{S}_{j^*}$
24: **end function**



(a) Before swapping. (b) After swapping.

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.

cluster upload traffic, then we 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 $\mathbb{S}$ (Lines 21-23).

Example: Figure 6 gives a swap example based on Figure 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 (Figure 6(a)) to two chunks (Figure 6(b)). The chunk H_2 will be re-organized in $\mathcal{U}$ and scheduled in next repair rounds.

C. 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,

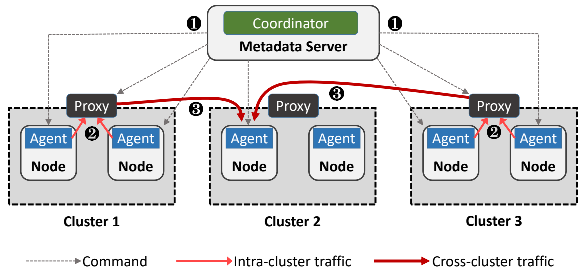


Fig. 7. System architecture of ClusterSR.

then the complexity of Algorithm 4 is $O(fl \log l)$. Algorithm 2 calls the SUBSTITUTE and SWAP functions for at most t times in each repair round, therefore the complexity of Algorithm 2 is $O(etfl \log l)$, where e is the number of repair rounds.

V. IMPLEMENTATION

We have implemented a ClusterSR prototype in C++ with around 2,700 lines of code. We use Jerasure v.1.2 [26] to realize the encoding and decoding functionalities.

System architecture: Figure 7 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 have the failed chunks and constructs repair solutions. It then issues the commands to the proxies and agents for instructing the repair procedure (step ❶ in Figure 7). 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 ❷). 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 destination node that is responsible for repairing the failed chunk (step ❸). 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 *no* modification to the HDFS codebase.

VI. Performance Evaluation

We carry out extensive performance evaluation, in terms of large-scale simulation and testbed experiments on Alibaba Cloud ECS. We expect to answer the following questions.

- Is ClusterSR effective on balancing the cross-cluster upload and download traffics for repair? (Experiment A.1)
- How sensitive ClusterSR is when any configuration varies? (Experiment A.2 & Experiment B.1-B.3)
- How much cross-cluster repair traffic can be reduced by ClusterSR? (Experiment A.2)
- How much computation time ClusterSR needs to establish the repair solution? (Experiment B.4)

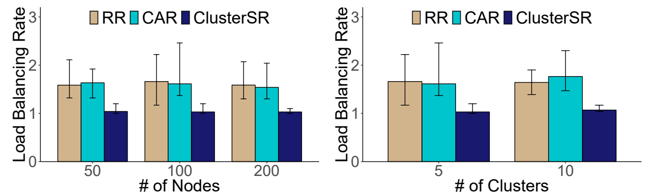
A. Large-Scale Simulation

We first 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 the cross-cluster repair traffic.

We compare ClusterSR with another two repair approaches, namely random repair (RR) and cross-rack-aware repair (CAR) [32]. 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.

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 [24]). 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 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).

[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) Varying number of nodes. (b) Varying number of clusters.

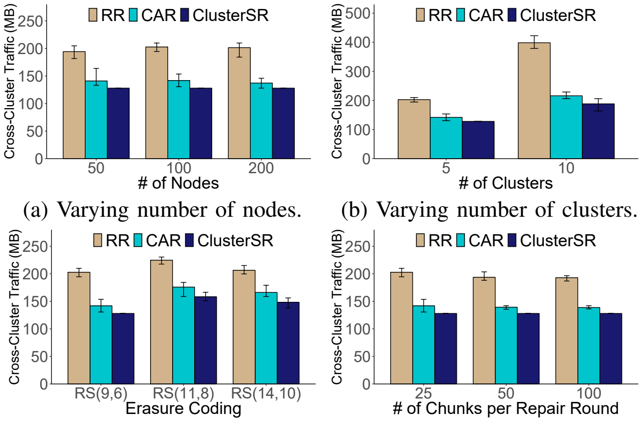
Fig. 8. Experiment A.1: load balancing test. The smaller value is better.

Experiment A.1 (Load balancing test): We first evaluate the load balancing rate (defined in Section III-C) in this test. Figure 8 shows the results when the numbers of nodes and clusters vary. We can derive 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 1.63 and 1.62, 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. Second, 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.

Experiment A.2 (Sensitivity test): In the sensitivity test, we vary one configuration while keeping other default configurations unchanged. We measure the average repair traffic (in unit of MBs) to be transferred across clusters for repairing a lost chunk. Figure 9 shows the results. First, ClusterSR incurs the least cross-cluster repair among the three repair approaches when the number of nodes (Figure 9(a)), the number of clusters (Figure 9(b)), the deployed erasure code (Figure 9(c)), and the number of chunks repaired in each round (Figure 9(d)) change. Specifically, ClusterSR can reduce 6.7%-12.8% and 28.0%-52.7% of the cross-cluster repair traffic when compared to CAR and RR, respectively. Second, we can notice that the cross-cluster repair traffic significantly increases with the number of clusters (Figure 9(b)). 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.

B. Testbed Experiments

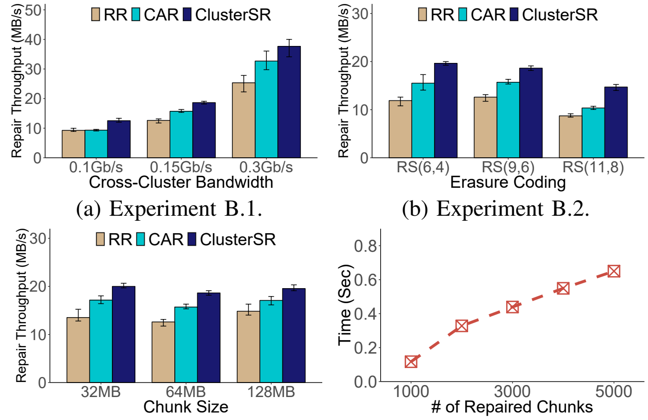
We further evaluate ClusterSR on Alibaba Cloud ECS [1] to study its performance in real-world cloud environment. We set up 21 virtual machine instances of type `ecs.g6.large` in East China region (Hangzhou Zone H). Each instance is equipped with 2 vCPUs (2.5 GHz Intel Xeon Platinum), 8GB



(a) Varying number of nodes. (b) Varying number of clusters.

(c) Varying erasure coding. (d) Varying chunks per round.

Fig. 9. Experiment A.2: sensitivity test. The smaller value is better.



(a) Experiment B.1. (b) Experiment B.2.

(c) Experiment B.3. (d) Experiment B.4.

Fig. 10. Experiments on Alibaba Cloud ECS.

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`).

Among the 21 instances, we deploy the ClusterSR 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 ClusterSR agents on four instances and deploy the ClusterSR 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 MB 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 the cross-cluster bandwidth): We first measure the repair throughput when the cross-cluster bandwidth is varied as 0.1 Gb/s, 0.15 Gb/s, and 0.3 Gb/s. Figure 10(a) 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% 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. Figure 10(b) shows the results. The repair throughput of all the three approaches decreases when the value k becomes large. 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% when compared with CAR and RR, respectively.

Experiment B.3 (Impact of the chunk size): We further evaluate the repair throughput under different chunk sizes, which are varied from 32 MB to 128 MB. Figure 10(c) shows the results. We can observe that 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% when compared with CAR and RR, respectively.

Experiment B.4 (Computation time): We finally use one instance to measure the computation time for ClusterSR to establish the repair solutions. We generate 200,000 stripes encoded by RS$(9, 6)$ and randomly distribute them in a cluster constructed by 100 nodes with five clusters. We suppose to repair 50 chunks in a repair round and vary the total number of chunks to be repaired. Figure 10(d) shows the results.

ClusterSR is extremely efficient to derive the repair solutions for repairing a large number of failed chunks. For example, it merely needs about 0.65 seconds to obtain the repair solutions for repairing 5,000 chunks. As ClusterSR only incurs extremely lightweight computation overhead, it is qualified to be deployed in the online repair scenario.

VII. Related Work

Repair-efficient codes. Some repair-efficient codes are designed to suppress the repair traffic. LRCs [15], [29] associate a subset of data chunks of a stripe with a local parity chunk, thereby trading additional storage for reduced repair traffic. Regenerating codes [8], [25] 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 propose to schedule the repair by fully utilizing the available bandwidth. PPR [20] decomposes a repair operation into partial repair sub-operations that are parallelized across multiple nodes. CAR [32] minimizes the cross-cluster repair traffic in data centers by accessing the minimum number of clusters in each chunk's repair. ECPipe [18] partitions a chunk into small-size slices and pipelines the repair of slices across nodes to achieve $O(1)$ repair time. DoubleR [14] performs both intra-cluster and cross-cluster regenerations to minimize the cross-cluster repair traffic in hierarchical data centers. These studies mainly focus on dedicated repair, while ClusterSR aims to balance the cross-cluster upload and download traffics in scattered repair.

Parity declustering. Parity declustering [12], [21] 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 [23], a replication-based storage system that scatters the replicas across the system for fast repair. FastPR [31] 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.

VIII. 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 further 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.

Acknowledgements: This work was supported by NSFC (61832011, 61602120, 61832020, 61702013, 61702569), National Key R&D Program of China (2017YFB1001600), special project of scientific and technological innovation strategy of Guangdong Province (2018B010109002), and Fujian Provincial Natural Science Foundation (2017J05102). Jiwu Shu is the corresponding author.

References

[1] Alibaba Cloud Elastic Compute Service. https://www.alibabacloud.com/product/ecs.
[2] Facebook HDFS. https://github.com/facebookarchive/hadoop-20, 2014.
[3] Apache Hadoop 3.1.1. https://hadoop.apache.org/docs/r3.1.1/, 2018.
[4] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar. Shufflewatcher: Shuffle-aware Scheduling in Multi-Tenant Mapreduce Clusters. In *Proc. of USENIX ATC*, 2014.
[5] Y. Chen, S. Mu, J. Li, C. Huang, J. Li, A. Ogus, and D. Phillips. Giza: Erasure Coding Objects across Global Data Centers. In *Proc. of USENIX ATC*, 2017.
[6] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, 2013.
[7] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *Proc. of USENIX NSDI*, 2016.
[8] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551, 2010.
[9] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.
[10] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, 2003.
[11] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, 2009.
[12] M. Holland, G. Gibson, and D. Siewiorek. Architectures and Algorithms for On-line Failure Recovery in Redundant Disk Arrays. *Distributed Parallel Databases*, 2(3):295–335, 1994.
[13] H. Hou, P. Lee, K. W. Shum, and Y. Hu. Rack-aware Regenerating Codes for Data Centers. *IEEE Transactions on Information Theory*, 65(8):4730–4745, 2019.
[14] Y. Hu, X. Li, M. Zhang, P. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice. *ACM Transactions on Storage*, 13(4):33, 2017.
[15] C. Huang, H. Simitci, Y. Xu, et al. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.
[16] H. Jin, C. Wu, X. Xie, J. Li, M. Guo, H. Lin, and J. Zhang. Approximate Code: A Cost-Effective Erasure Coding Framework for Tiered Video Storage in Cloud Systems. In *Proc. of ACM ICPP*, 2019.
[17] 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. of USENIX FAST*, 2012.
[18] R. Li, X. Li, P. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*, 2017.
[19] H. Liu, M. Mukerjee, C. Li, et al. Scheduling Techniques for Hybrid Circuit/Packet Networks. In *Proc. of ACM CoNEXT*, 2015.
[20] S. Mitra, R. Panta, M. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of ACM EuroSys*, 2016.
[21] R. Muntz and J. Lui. Performance Analysis of Disk Arrays under Failure. In *Proc. of VLDB*, Aug 1990.
[22] S. Muralidhar, W. Lloyd, S. Roy, et al. F4: Facebook's Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.
[23] D. Ongaro, S. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, 2011.
[24] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proceedings of the VLDB Endowment*, 6(11):1092–1101, 2013.
[25] L. Pamies-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. of USENIX FAST*, 2016.
[26] J. Plank, S. Simmerman, and C. Schuman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.
[27] K. Rashmi, N. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proc. of ACM SIGCOMM*, 2015.
[28] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.
[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment*, 6(5):325–336, 2013.
[30] Z. Shen and P. Lee. Cross-Rack-Aware Updates in Erasure-Coded Data Centers. In *Proc. of ACM ICPP*, 2018.
[31] Z. Shen, X. Li, and P. Lee. Fast Predictive Repair in Erasure-Coded Storage. In *Proc. of IEEE/IFIP DSN*, 2019.
[32] Z. Shen, J. Shu, and P. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2016.
[33] I. Tamo and A. Barg. A Family of Optimal Locally Recoverable Codes. *IEEE Transactions on Information Theory*, 60(8):4661–4676, 2014.
[34] H. Weatherspoon and J. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, 2002.