2023 IEEE International Conference on Cluster Computing (CLUSTER) | 979-8-3503-0792-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/CLUSTER52292.2023.00017

FullRepair: Towards Optimal Repair Pipelining in Erasure-Coded Clustered Storage Systems

Yuzuo Zhang, Xinyuan Tu, Lin Wang, Yuchong Hu, Fang Wang, Ye Wang
Huazhong University of Science & Technology, Wuhan, China
Email: {jadezhang, xytu, wanglin2021, yuchonghu, wangfang, u202013878}@hust.edu.cn

***Abstract*—Clustered storage systems often deploy erasure coding that encodes data into coded chunks and distributes them across nodes to tolerate node failures. It is a storage-efficient redundancy scheme but incurs high repair penalty; thus some state-of-the-arts aim to pipeline the above repair process to improve the repair performance. However, we observe that all existing repair pipelining methods only use a single pipeline, making network bandwidth resources of storage nodes underutilized.**

In this paper, we propose FullRepair, a new repair pipelining mechanism based on multiple pipelines with the aim of fully exploiting all available bandwidth resources during repair. We construct four constraints to model the repair pipelining problem such that we can obtain the optimal pipelined repair throughput under full bandwidth utilization. We design a multi-pipeline scheduling scheme for FullRepair so as to achieve the above optimality. Experiments on the Amazon EC2 show that compared with the state-of-the-art repair pipelining methods RP and PivotRepair, FullRepair reduces the repair time of single chunk by up to 45.40% and 33.19%, respectively.

***Index Terms*—distributed storage, erasure coding, parallelism, data reliability, recovery**

I. Introduction

A clustered storage system enables businesses to manage the access of big data across multiple storage nodes, and it is often composed of commodity servers which are inexpensive and thus have high chance of failures, so data redundancy is often added against failures to maintain data reliability for fault tolerance. Compared to the traditional redundancy strategy – replication [1], *erasure coding* is favored by many clustered storage systems [2]–[4] as it can provide the same fault tolerance with less storage overhead. Erasure coding encodes k fixed-size data units, called *data chunks*, of the origin data into $n-k$ ($n>k$) *parity chunks*, satisfying that the original k data chunks can be rebuilt from any k out of the n data and parity chunks. In this way, a clustered storage system can tolerate up to $n-k$ node failures simultaneously by storing the n chunks in different n nodes [2], [5], [6].

While erasure coding provides low-cost redundancy, it incurs high repair penalty. Specifically, if one data chunk fails, the node that requests the data chunk (called *requester*) needs to repair it by retrieving k chunks from other k available nodes (called *helpers*) and decoding the requested chunk; then the repair penalty means that the erasure-coded single-chunk repair task has to transmit k times the data of the replication-based single-chunk repair task. Here, we call the k transmitted chunks during repair *repair bandwidth*. Thus, how to efficiently handle the single-chunk repair task is one of the most critical issues in erasure-coded clustered storage systems.

Further, in the storage cluster, many foreground jobs will incur network congestion [7]–[9] with the repair task. Thus, many studies aim to make better use of the nodes' *available bandwidth* to speed up the repair task [2], [9]–[13]. Among them, some state-of-the-art studies (e.g., RP [12], PPT [13], and PivotRepair [9]) leverage *repair pipelining* to distribute the repair bandwidth (i.e., the k transmitted chunks) smartly across nodes to mitigate network congestion, thereby improve the erasure-coded repair performance. For example, RP [12] uses sub-chunks to pipeline the repair across helpers to evenly distribute the repair bandwidth; PPT [13] and PivotRepair [9] use a tree-like path to pipeline the repair under heterogeneous network environments.

However, our observation (§II-C) shows that the above repair pipelining studies (RP [12], PPT [13], and PivotRepair [9]) are all based on one single pipeline, which leads to underutilizing the nodes' available bandwidth. Specifically, when one chunk fails due to one failed node, the existing single-pipeline studies only pipeline the repair of the failed chunk via decoding it from the k available chunks, which are retrieved from k nodes chosen out of $n-1$ non-failed nodes; yet we find that there remain $n-1-k$ non-failed nodes' whose available bandwidth are still not used during repair. In view of this, we pose the following question: *Can we exploit all the $n-1$ non-failed nodes' available bandwidth to repair the failed chunk so as to further improve the repair performance?*

In this paper, we propose FullRepair, whose main idea is to use multiple pipelines instead of single pipeline such that the entire available bandwidth can be fully leveraged during repair. Our contributions include:

- We show via observations that in storage clusters, there remain many available bandwidth resources when using existing repair pipelining methods, as they only use single pipeline and not all the non-failed nodes' available bandwidth are fully utilized (§II).
- We pose and analyze the multi-pipeline repair problem, and obtain the optimal pipelined repair throughput (§III). We also design a greedy algorithm for FullRepair that achieves the optimality via making full use of multiple pipelines of all available bandwidth resources during repair (§IV).
- We prototype and evaluate FullRepair on Amazon EC2. Compared to RP [12], PPT [13] and PivotRepair [9], FullRepair reduces the repair time of a single-chunk repair by up to 45.4%,

62.93% and 33.19%, respectively (§V). Our prototype is open-sourced at: https://github.com/YuchongHu/FullRepair.

II. Background and Motivation

A. Basics of Erasure Coding

Erasure coding is popular in the literature with various kinds of constructions [14]. Among them, Reed-Solomon (RS) [15] codes are mostly adopted by practical clustered storage systems (e.g., Swift [16], QFS [17], HDFS [6], and Ceph [3]). A RS code is often configured with two parameters n and k ($n > k$), and an (n, k) RS code encodes k fixed-size *chunks* of original data (*data chunks*) into $n-k$ encoded chunks with the same size (*parity chunks*), such that any k chunks of the n chunks can rebuild the original k data chunks. The set of all the n chunks is called a *stripe*. This paper focuses on RS codes.

In an (n,k) RS-coded stripe, its $n-k$ parity chunks (denoted by P_j, $1 \le j \le n-k$) are linear combinations of its k data chunks (denoted by D_i, $1 \le i \le k$), represented as $P_j = \sum_{i=1}^{k} \alpha_j^{i-1} D_i$, where α_j^{i-1} represent the encoding coefficients. The additions and multiplications are based on the Galois Field GF(2^w) [15] over w-bit words, meaning that the addition between two chunks is performed via bitwise-XORs and the multiplication between a chunk and a constant is performed via multiplying every w-bit word of the chunk by the constant.

Since the above encoding process is a set of linear combinations, the repair of a failed chunk is also a set of linear combinations. Taking $(5,3)$ RS code as an example, its first parity chunk $P_1 = D_1 + \alpha_1 D_2 + \alpha_1^2 D_3$, so its first data chunk D_1 can be repaired by

$$D_1 = P_1 + \alpha_1 D_2 + \alpha_1^2 D_3. \quad (1)$$

Due to the linearity of Equation (1), the repair process of a single chunk is capable of being pipelined, specified in §II-B.

B. Pipelined Repair of Erasure Coding

Erasure coding is known to its repair penalty [2], [18], [19], which means that it incurs more data transfers than the size of failed data to be repaired. For example, suppose that a $(5,3)$ RS code stores its five chunks D_1, D_2, D_3, P_1 and P_2 in five nodes N_1, N_2, N_3, N_4, and N_5, respectively. When node N_1 fails, its chunk D_1 is unavailable. Based on Equation (1), the requester (denoted by R) needs to download three chunks D_2, D_3 and P_1 (or P_2) from three corresponding *helpers* N_2, N_3 and N_4 to repair the requested chunk D_1, which indicates the repair penalty.

Conventionally, the above repair penalty also leads to the congestion at the requestor R and increases the overall repair time. Fig. 1(a) illustrates that the requester R downloads one chunk from each of the three helpers N_2, N_3 and N_4, so the downlink of the requestor R is three times more congested than each of the helpers N_2, N_3 and N_4.

To address congestion in erasure-coded repair, recent studies propose repair pipelining schemes as follows.

Repair Pipelining (RP) [12]: RP splits every data chunk into slices, and pipelines them through a chain-like path, such that all the links in the pipeline transfer the same amount of data.

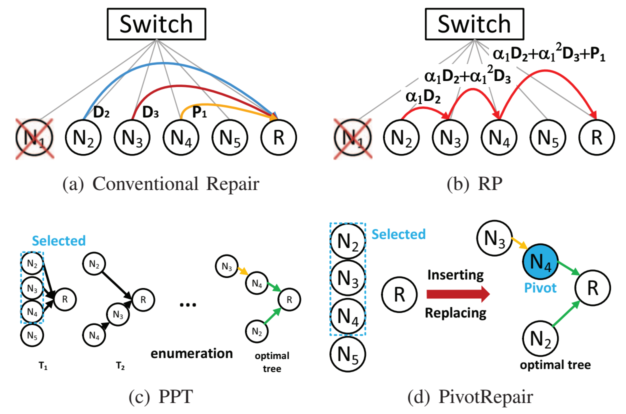


Figure 1: Examples of the conventional repair, RP, PPT and PivotRepair when repairing a failed chunk.

Fig. 1(b) shows that to repair D_1 based on Equation (1), the three links transmit $\alpha_1 D_2$, $\alpha_1 D_2 + \alpha_1^2 D_3$ and $\alpha_1 D_2 + \alpha_1^2 D_3 + P_1$, all of which have equal size, since the XOR-based additions ensure that the addition results have the same size as the original chunks.

Parallel Pipeline Tree (PPT) [13]: PPT pipelines the repair via a tree-like path rather than a chain-like path, such that it outperforms RP for heterogeneous network, which may bottleneck the repair performance by the slowest link of the chain-like path. PPT emulates all possible pipeline trees and selects the optimal one, as illustrated in Fig. 1(c).

PivotRepair [9]: PivotRepair accelerates the construction of the tree-like path of PPT via inserting uncongested nodes (called *pivot*) into the tree to quickly initialize the tree while replacing the congested nodes to bypass the congestion. Therefore, PivotRepair can significantly reduce the time spent by the emulation of PPT, as illustrated in Fig. 1(d).

Note that the above repair pipelining studies (RP [12], PPT [13], and PivotRepair [9]) are all based on one single pipeline: RP uses a single chain-like pipeline, and both PPT and PivotRepair use a single tree-like pipeline; more importantly, there are only k helpers involved to construct the single pipeline (see Fig. 1(b), 1(c) and 1(d)). It means that when one node fails, existing repair pipelining methods only use k out of $n-1$ non-failed nodes, such that there remain $n-1-k$ non-failed nodes that are not involved in the pipelined repair, thus making these nodes' available repair bandwidth underutilized. For example, the node N_5 never provides its bandwidth resources for repairing the failed chunk D_1 in Fig. 1(b), 1(c) and 1(d).

Therefore, it is natural to ask whether it is helpful to fully use all available repair bandwidth of the $n-1$ non-failed nodes (e.g., N_2, N_3, N_4 and N_5 in Fig. 1) when pipelining (n,k) RS-coded single-chunk repairs. To this end, we will discuss the benefit based on observations in §II-C.

C. Observation

To understand the benefit of fully using all available repair bandwidth resources for RS coded pipelined repair, we study the bandwidth details of practical clustered storage workloads similar to PivotRepair [9].

We re-run the traces in PivotRepair [9], including TPC-DS [20], TPC-H [21] and SWIM [22]. Here, TPC-DS is a comprehensive benchmark for databases and big-data systems supporting SQL; TPC-H is a classical benchmark for decision support of business databases; SWIM is a MapReduce trace on a cluster of 3000 machines at Facebook within 1.5 months. We re-construct a clustered storage system in PivotRepair [9], where there are 16 storage nodes and the total bandwidth of each node is 1 Gbps (shared by foreground jobs and repair tasks, which are limited in the system using the Linux tc command), while the available repair bandwidth for repair tasks (defined as *available repair bandwidth*) is measured using the Linux nload command. We generate 6000 sets of bandwidth data in each of the three workloads.

We analyze the available repair bandwidth resources at each node under the above three workloads, and compare the amount of bandwidth actually used by the pipelined repair of the three state-of-the-art algorithms: RP, PPT and PivotRepair. Similar to PivotRepair [9], we use the coefficient of variation (coefficient of variation, i.e., the ratio of the standard deviation to the mean) of the average node bandwidth used by the 16 nodes in each second to indicate the degree of unevenness of the network bandwidth (e.g., $C_v = 0$ means that all nodes use the same bandwidth).

Based on the above experiments, we observe the results shown in Table I, where the distribution of bandwidth resources indicates the actual use of bandwidth resources in each algorithm. The *selected nodes' used bandwidth ratio* is the ratio of used repair bandwidth of the nodes that are selected for repair (i.e. helpers) to the entire available repair bandwidth resources (we define it as the *bandwidth utilization* of the algorithm), the *unselected nodes' bandwidth ratio* is the ratio of bandwidth of the nodes that are not selected as helper to the entire available repair bandwidth resources, and the *selected nodes' unused bandwidth ratio* is the ratio of the remaining bandwidth resources of the selected helpers to the entire available repair bandwidth resources.

Table I shows that none of the existing algorithms fully utilizes the bandwidth resources of all nodes for different network bandwidth scenarios. That is, the selected nodes' used bandwidth ratios (i.e., the bandwidth utilization) are far from 100%, which shows the benefit of further exploiting the available bandwidth resources of pipelined repair.

Specifically, the distribution of bandwidth resources for RP, PPT, and PivotRepair is the same when the network bandwidth is even (i.e., when the $0 \leq C_v < 0.1$, the values are small) because all the three schemes will follow the chain-like path to pipeline the repair; the bandwidth utilization is also relatively high at this time (all above 70%). However, as the bandwidth unevenness of the network increases (i.e., the C_v value increases), the bandwidth utilization of all algorithms decreases. For example, when $0.4 \leq C_v < 0.5$, the bandwidth resource utilization of PPT and PivotRepair is 39.4%, while that of RP is only 29.5%.

Table I: Distribution of network bandwidth resources for RP, PPT and PivotRepair in networks with different degree of unevenness of the network bandwidth.

Bandwidth Inequality	Parallel Repair Algorithm	Distribution of Network Bandwidth Resource		
		Selected nodes' used bandwidth ratio	Unselected nodes' bandwidth ratio	Selected nodes' unused bandwidth ratio
$0 \leq C_v < 0.1$	RP	76.5%	19.0%	4.5%
	PPT PivotRepair	76.5%	19.0%	4.5%
$0.1 \leq C_v < 0.2$	RP	73.2%	16.8%	10.0%
	PPT PivotRepair	73.2%	16.8%	10.0%
$0.2 \leq C_v < 0.3$	RP	70.5%	14.9%	14.6%
	PPT PivotRepair	70.6%	14.9%	14.6%
$0.3 \leq C_v < 0.4$	RP	46.1%	11.9%	42.0%
	PPT PivotRepair	47.3%	12.0%	40.7%
$0.4 \leq C_v < 0.5$	RP	29.5%	10.6%	59.9%
	PPT PivotRepair	39.4%	11.7%	48.9%

More importantly, even if the bandwidth utilization of each algorithm is high when the network bandwidth is even, there are still a lot of unused bandwidth resources in the unselected nodes. For example, when $C_v < 0.3$, the unselected nodes' bandwidth ratio accounts for more than 14.9% in different algorithms. In addition, when the network bandwidth is not even, even if the unselected nodes' bandwidth ratio decreases simultaneously, there remain a lot of available repair bandwidth resources of selected nodes that are not used for repair. For example, when $C_v \geq 0.3$, the selected nodes' unused bandwidth ratio accounts for more than 42.0% in different algorithms.

D. Motivation

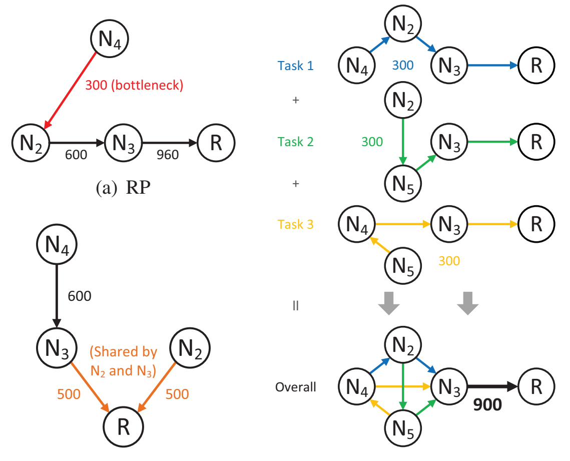
Based on the above observations, we have the following conclusions:

Conclusion 1: When the network condition is even, the bandwidth utilization is relatively high, but there still remain many available repair bandwidth resources in the unselected nodes. Therefore, if all $n-1$ non-failed nodes in the network are fully utilized, instead of selecting only k nodes of existing pipelined repair algorithms, the bandwidth utilization can be increased further.

Conclusion 2: When the network condition is not even, the bandwidth utilization is relatively low. It is because all existing pipelined repair algorithms only enable a single pipeline, so some bandwidth-insufficient selected nodes along the single pipeline will make some bandwidth-sufficient selected nodes of the pipeline not take full advantages of their bandwidth resources. Therefore, if we can enable multiple pipelines, instead of single one pipeline, then the bandwidth-sufficient selected nodes will further provide their bandwidth resources, such that the bandwidth utilization can be increased further.

Main idea: Based on the Conclusions 1 and 2, our main idea is (1) to make full use of the available repair bandwidth of all

	N_2	N_3	N_4	N_5	R
Available Uplink Bandwidth (Mbps)	600	960	600	600	1000
Available Downlink Bandwidth (Mbps)	300	1000	300	300	1000



(a) RP

(b) PPT and PivotRepair

(c) Our Design

Figure 2: Example of a $(5,3)$ RS code repairing. The table in the figure shows the uplink and downlink bandwidths of each node (including the requester) in the example, and the diagrams below depict the pipelines constructed by different algorithms. In the diagrams, the colored lines and corresponding numbers represent the final transmission speed (in Mbps) of the pipeline.

$n-1$ non-failed nodes, and (2) to construct multiple pipelines to perform the repair, which will be motivated in §II-E.

E. Motivating Example

We use Fig. 2 as an example to motivate our main idea. We consider the same setting in Fig. 1, where the requester R needs to repair the failed chunk from four non-failed nodes N_2, N_3, N_4 and N_5. Here, the four non-failed nodes have different available uplink/downlink bandwidth resources during repair since they have different foreground jobs.

RP would select three nodes with more sufficient bandwidth: N_2, N_4, and N_5 to be the helpers to form its chain-like pipeline, whose performance is bottlenecked by the slowest downlink of N_2, such that the pipeline throughput is limited to N_2's downlink available bandwidth: 300 Mbps, as shown in Fig2(a). PPT and PivotRepair, can bypass congested downlinks by letting N_3 and N_2 share their sufficient downlink bandwidth to R, so the pipeline throughput can reach 500 Mbps, as shown in Fig. 2(b). However, PPT and PivotRepair still waste many bandwidth resources of the network, such as N_5's entire bandwidth, N_2's donwlink bandwidth (300 Mpbs unused), N_3's donwlink bandwidth (500 Mpbs unused), and N_4's donwlink bandwidth (300 Mpbs unused).

Based on the main idea proposed in §II-D, we construct three pipelines that can fully exploit the bandwidth resources of the network. In this example, we let each of the downlinks of N_2, N_4, and N_5 supports one pipeline and use these three pipelines to share the sufficient bandwidth of N_3 and R, as illustrated in Fig. 2(c). In this way, each of the three pipelines can have a throughput of 300 Mbps and thus the entire pipeline throughput can reach up to 900 Mbps, which is much higher than that of RP, PPT and PivotRepair.

Based on the insight of this example, we propose a new repair scheme which can leverage multiple pipelines to fully exploit the available bandwidth of all non-failed nodes, called FullRepair, which will be analyzed and designed in §III and §IV, respectively.

III. Analysis

A. Definitions

To analyze how FullRepair designs multiple pipelines to exploit all available bandwidths in theory, we first give the following definitions:

Pipelined repair throughput of FullRepair: We define the sum of all the pipelines' throughput as the pipelined repair throughput of FullRepair.

Ideal pipelined repair state of FullRepair: It is assumed that the repaired data can be equally divided into an infinite number of slices and infinite tasks can be performed simultaneously in each link without affecting each other. In this ideal condition, all the available network bandwidth resources can be fully exhausted by an infinite number of slice-based pipelines.

Ideal pipelined repair throughput of FullRepair: In the ideal pipelined repair state, the pipelined repair throughput of FullRepair can be maximum, which we call *ideal pipelined repair throughput* of FullRepair, denoted by *IdealThroughput*.

Ideal uplink/downlink bandwidth of FullRepair: In the ideal pipelined repair state, each nodes' available uplink/downlink bandwidth can be fully exhausted, which we call *ideal uplink/downlink bandwidth*, denoted by *IdealUplink* and *IdealDownlink*.

B. Constraints

Based on the observations (§II-C) of real network bandwidth, we describe the relationship between the ideal pipelined repair throughput and ideal uplink/downlink bandwidth via four constraints, described as follows.

(1) The uplink constraint: *The available bandwidths of a network's uplinks limit the pipelined repair throughput.*

$$IdealThroughput \leq (\sum_{i=1}^{n} IdealUplink_i)/k \tag{2}$$

An (n,k) RS code requires to transmit k times the data that is needed to be rebuilt. In other words, a requester can only rebuild $1/k$ of the size of transmission in the network. Thus, the pipelined repair throughput cannot be larger than $1/k$ of the whole network's upload speed.

(2) The downlink constraint: *The available bandwidths of a network's downlinks limit the pipelined repair throughput.*

$$IdealThroughput \leq (\sum_{i=0}^{n} IdealDownlink_i)/k \tag{3}$$

Similar to (2), the pipelined repair throughput cannot be larger than $1/k$ of the whole network's download speed.

(3) The storage constraint: *A node can only supply the data that it stores.*

$$IdealThroughput \geq \max_{1 \leq i \leq n} \{IdealUplink_i\} \quad (4)$$

During the repair procedure, every node's upload speed can never be larger than the speed of the rebuilding.

(4) The repairing constraint: *A node cannot download the data of the pipelines that the node does not participate in.* $\forall i \in [1,n]$:

$$IdealDownlink_i \leq IdealUplink_i * (k-1) \quad (5)$$

As every helper only helps the requester to rebuild data and never keeps data during repairing, all the pipelines finally flow into the requester; that is, a helper should send all the downloaded data out after computing its own data.

C. Optimality

We will prove that all the above four constraints (inequalities) hold when FullRepair achieves the optimality, i.e., in the ideal pipelined repair state (Theorem 1).

Theorem 1. *The four Equations* (2), (3), (4) *and* (5) *hold when the pipelined repair is in the ideal pipelined repair state.*

Proof: In the ideal repair state, the amount of data transmitted by any node is just the amount of data provided by the node itself.

For Equation (2), since repairing each slice of data requires k slices of data to participate in, if the amount of data provided by each node is x, and the final amount of repaired data is y, then we have $\sum_{x=i}^{n} x_i = ky$. In the ideal pipelined repair state, the transmission and restoration are carried out synchronously, which means that $\sum_{x=i}^{n} \frac{x_i}{t} = k\frac{y}{t}$. Thus, the sum of the average upload throughput of each node is k times the ideal pipelined repair throughput. Equation (2) holds.

For Equation (3), it is clear that the ideal pipelined repair throughput is $\frac{1}{k}$ of the sum of all ideal uplink bandwidths based on Equation (2). During the whole repair process, the total data uploaded is equal to the total data downloaded. Therefore, the sum of average download throughput is equal to the sum of average download speeds. Equation (3) holds.

For Equation (4), it is clear that each repaired slice of data is reconstructed from different k nodes. Assume that to repair a failed slice b_i, we need to retrieve k slices a_{i1}, a_{i1}, ...,a_{ik} from k nodes N_{i1}, N_{i2}, ..., N_{ik}, respectively. Since every node can only provide one slice of data to the other node at most, the total amount of data provided by each node cannot exceed the total amount of final repaired data; that is, the average upload speed of each node cannot be greater than the data repair speed. Equation (4) holds.

For Equation (5), we assume that each part of the data uploaded by the node N_i is b_i, and then we have $b_i = a_{0i} + \sum_{j=i}^{x_i} a_{ji}$, where a_{0i} is the data provided by itself, and a_{ji} is the data received from other nodes. Because each slice needs k available slices for repair, we have $0 \leq x_i \leq k-1$. Since a_{0i} and a_{ji} have the same size, we have $y_i \leq (k-1)x_i$. Thus, we can have $\sum y_i \leq (k-1)\sum x_i$, which means that the amount of data received by a node is less than or equal to the $k-1$ times of the data provided by itself. Equation (5) holds. ∎

Therefore, we can obtain the maximum (i.e., ideal) pipelined repair throughput based on Theorem 1 to design optimal repair pipelining algorithm, which will be specified in §IV.

IV. FullRepair Design

A. Calculating Maximum Pipelined Repair Throughput

Design idea: Based on Theorem 1, we will use its four inequalities to calculate the maximum pipelined repair throughput t_{max} from the available bandwidth of each node. Obviously, the pipelined repair throughput t_{max} is upper bounded by the maximum pipelined repair throughput that can be achieved under the available network bandwidth.

Design details: In Algorithm 1, U_i and D_i denote the bandwidths of N_i's uplink and downlink, respectively, and the whole procedure can be divided into two parts: limiting the pipelined repair throughput by uplinks (Lines 2-12) and limiting the pipelined repair throughput by downlinks (Lines 13-25).

(1) Limiting by uplinks: The uplinks' available bandwidths limit the pipelined repair throughput through Equations (2) and (4), and all the nodes except the requester are involved in the procedure. We use an empty list E for collecting the nodes which have violated Equation (4) (Line 2), and a list of the nodes that has not been picked is also required, so E initially contains all the helpers (Line 3). When the number of the picked nodes increases to $k-1$, Equation (4) must have been satisfied, so after repeating the pick operation in Lines 4-8 several times, the upload bandwidth resources will satisfy both Equations (2) and (4). Since the maximum pipelined repair throughput cannot be larger than the downlink available bandwidth of the requester, the smaller one of the upload-limited throughput and D_0 is the result (Line 9). Finally, the uplink bandwidths of the unselected nodes in E should be set equal to the maximum pipelined repair throughput (Lines 10-12).

(2) Limiting by downlinks: The limitation of download links is mainly based on Equations (3) and (5), but it would also be influenced by the limitation of (4), since they have the same definition IdealUplink. However, this only happens when downlink-limited pipelined repair throughput is smaller than the uplink-limited one, so that the limitation of Equation (4) would not let the uplink-limited throughput be smaller than the downlink-limited throughput. Due to the mutual influences of Equations (5), (3) and (4), the maximum pipelined repair throughput will be alternately limited until it is stable (Lines 13-25). After limiting the pipelined repair throughput by downlinks, the algorithm finishes as the result satisfies all the four constraints in Theorem 1.

Design example: Take the bandwidth condition in Fig. 2 as an example. The calculation results of each node is shown in the table II. The sum of the available uplink bandwidths is 2760 Mbps (excluding the requester R). As no node has been picked out, the current pipelined repair throughput obtained from Equation (2) is the result of (sum$/k$), i.e., 920 Mbps in the example. Since the current pipelined repair throughput is

Algorithm 1 Maximum Pipelined Repair Throughput Calculation

Input: bandwidths of uplink U and downlink D, n, k
Output: the maximum pipelined repair throughput t_{max}

```
 1: procedure MAIN
 2:     $E$ is an empty list
 3:     $L$ is a list includes numbers $\in [1,n]$
 4:     while $(\sum U_{i,i\in L})/(k-E.\text{length}) < \max U_{i,i\in L}$ do
 5:         $i \leftarrow$the number in $L$ which makes the max $U_i$
 6:         $L$.remove($i$)
 7:         $E$.add($i$)
 8:     end while
 9:     $c \leftarrow \min\{(\sum U_{i,i\in L})/(k-E.\text{length}), D_0\}$
10:     for all $i \in E$ do
11:         $U_i \leftarrow c$
12:     end for
13:     repeat
14:         $c \leftarrow \min\{(\sum_{i=0}^{n} D_i)/k,\ c\}$
15:         $flag \leftarrow$ True
16:         for each number $i \in [1,n]$ do
17:             $U_i \leftarrow \min\{c, U_i\}$
18:             if $U_i * (k-1) < D_i$ then
19:                 $D_i \leftarrow U_i * (k-1)$
20:                 $flag \leftarrow$ False
21:             end if
22:         end for
23:     until $flag$
24:     return $t_{max}$
25: end procedure
```

smaller than the maximum uplink bandwidth among the nodes, N_3 should be picked out. It makes the sum change to 1800 Mbps and the expression of the pipelined repair throughput be $(\text{sum}/(k-1))$, and the result becomes 900 Mbps. At this time, the pipelined repair throughput is larger than all the uplink bandwidths in L, which contains N_2, N_4, and N_5. Therefore, after calculating the limitation of uplinks, the pipelined repair throughput is 900 Mbps and the uplink of N_3 is also changed to 900 Mbps. However, the sum of the downlink bandwidths includes the downlink of R, which comes to be 2900 Mbps in the example, so the pipelined repair throughput, limited by Equation (3), turns out to be 2900/3 Mbps, which is larger than the pipelined repair throughput calculated from uplinks. As every node has already satisfied Equations (3) and (5), no further operation should be done. Thus, the maximum network pipelined repair throughput is 900 Mbps, which means the solution in the figure is one of the best schedules that can be found.

B. Scheduling Pipelined Repair Tasks

Design idea: The t_{max} calculated by Algorithm 1 is the maximum pipelined repair throughput that can be achieved in the current network environment. In order to achieve this repair throughput, we design a greedy algorithm (Algorithm 2) to schedule the tasks of each pipeline, so that the sum of each pipeline throughput can reach the value of t_{max}.

Table II: Example of Maximum Pipelined Repair Throughput Calculation

Node		N2	N3	N4	N5	R
Available Uplink Bandwidth (Mbps)	Before Algorithm 1	600	960	600	600	\
	After Algorithm 1	600	900	600	600	\
Available Downlink Bandwidth (Mbps)	Before Algorithm 1	300	1000	300	300	1000
	After Algorithm 1	300	1000	300	300	1000

Algorithm 2 Pipelined Repair Task Scheduling

Input: the set of helpers H, requester R, maximum pipelined repair throughput t_{max}
Output: list of the pipelines L

```
 1: procedure MAIN
 2:     Sort $H$ by descending downlink
 3:     $remain \leftarrow c$
 4:     for each $node \in H$ do
 5:         $speed \leftarrow \min\{remain, node.\text{download}/(k-1)\}$
 6:         $node$.task.speed $\leftarrow speed$
 7:         $remain \leftarrow remain - speed$
 8:     end for
 9:     if $remain > 0$ then
10:         $R$.task.speed $\leftarrow remain$
11:     end if
12:     Sort $H$ by descending (uplink $-$ task.speed)
13:     Numbering the tasks by the order of $H$
14:     $T_{unassigned}$ contains all the tasks sorted by descending
        (task.remian, task.id)
15:     for each $task \in T_{unassigned}$ do
16:         $task$.remain $\leftarrow k$
17:     end for
18:     $T_{assigned}$ is an empty task list sorted by descending
        (task.remain, 0-task.id)
19:     for each $node \in H$ do
20:         TASKASSIGN($node$, $T_{unassigned}$, $T_{assigned}$)
21:     end for
22: end procedure
```

Design details: In Algorithm 2, as the pipeline tasks are divided and assigned to different nodes with different nodes' bandwidth, each pipeline will have a node as the *hub* that collects the data slices from the remaining $k-1$ nodes in the pipeline and sends the calculated result to the requester with its local data slices. We call the pipeline task that each node participates in as the hub *own task*, and the pipeline task that each node sending out data slices the *sending task*. Algorithm 2 is divided into two parts: assign the nodes' bandwidth for own tasks (Lines 2-11) and assign the nodes' bandwidth for sending tasks (Lines 12-21). The sending task assignment includes a function TASKASSIGN which specifies the assigning process.

(1) *Own task assignment:*

First, all nodes are sorted in descending order according to their available downlink bandwidth (denoted by $Downlink_i$) calculated from Algorithm 1 (Line 2). Since the total throughput

```
1:  function TASKASSIGN($node$,$T_{unassigned}$, $T_{assigned}$)
2:      if $node$.task.size $> 0$ then
3:          $T_{unassigned}$[node.task.id].remain - = 1
4:          Move the task from $T_{unassigned}$ into $T_{assigned}$
5:          $node$.uplink - = $node$.task.speed
6:      end if
7:      while any task in $T_{assigned}$ or $T_{unassigned}$ can be assigned to $node$ do
8:          $task\_assign \leftarrow$ first task in $T_{assigned}$ that can be assigned
9:          $t \leftarrow$ first task in $T_{unassigned}$ that can be assigned
10:         if $t$.remain $>$ $task\_assign$.remain then
11:             $task\_assign \leftarrow t$
12:         end if
13:         Assign $task_{assign}$ to $node$
14:     end while
15:     if $node$.uplink $> 0$ AND $T_{assigned}[0]$.remain $> 0$ then
16:         Find the $task$ in $T_{unassigned}$ that can be added to $node$ and exchange
17:     end if
18: end function
```

of all pipelines is the maximum pipelined repair throughput t_{max}, the total throughput of the assigned own tasks is also t_{max} (Line 3). As all nodes need to receive $k-1$ nodes' data slices in their own tasks, so the maximum value of each node's own task throughput is $\frac{Downlink_i}{k-1}$, which will be assigned as their own tasks throughput in the descending order unless the left total throughput is not enough (Lines 4-8). If there are some left pipeline throughput, they will be assigned as node R's own tasks (Lines 9-11).

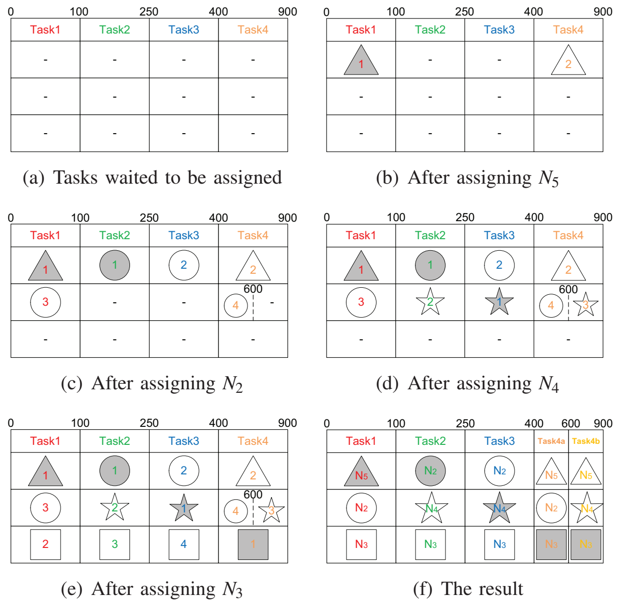
(2)*Sending task assignment:*

All pipelines have been established after the assignment of own tasks. Then the algorithm only needs to pair the remaining bandwidth of each node (as sending tasks) with the own task of other nodes.

First, the algorithm sorts all nodes in descending order according to the unused available uplink bandwidth and numbers each pipeline task in this order (Lines 12-13). Then, all pipeline tasks (own tasks and sending tasks) are stored in the queue $T_{unassigned}$ in descending order according to the number of unassigned sending tasks and the pipeline tasks' number (Lines 14-17). When a pipeline's sending task is assigned, it will be transferred to the queue $T_{assigned}$ and sorted (Line 18). Each node is assigned sending tasks in descending order according to the unused available uplink bandwidth (Lines 19-21). The specific assignment process is described in following function *TASKASSIGN*.

The function *TASKASSIGN* divides the assignment into two types:

- *Regular sending task assignment*: If a node has an own task, it first assigns the corresponding sending tasks for the node's own task (Lines 2-6). After assigning the sending portion of its own task, if there still remain uplink bandwidth resources unused for that node and there are still sending tasks in $T_{assigned}$ (or $T_{unassigned}$) that can be allocated, it will continue to allocate sending tasks to it until the sending tasks in the queues are all assigned (Lines 7-14).
- *Task exchange*: If a node still has unused uplink bandwidth resources after regular sending task assignment and there are still sending tasks in $T_{assigned}$, it is possible to swap the current node that needs to be allocated with one of the nodes that has completed its allocation. In this way, the assignment of tasks in $T_{assigned}$ can be performed while $T_{unassigned}$ remains unchanged (Lines 15-17).



0
100
250
400
900
600
Task1
Task2
Task3
Task4
Task4a
Task4b
N5
N2
N4
N3
(a) Tasks waited to be assigned
(b) After assigning N_5
(c) After assigning N_2
(d) After assigning N_4
(e) After assigning N_3
(f) The result

Figure 3: Illustration of running Algorithm 2:The figure depicts the process of task assignment. Each task occupies a grid, the tasks assigned to different node will be marked with different shapes(circles for N_2, squares for N_3, pentagram for N_4 and triangle for N_5). Besides, the shadowed ones represent noeds' own tasks and the white ones represents nodes' sending tasks. The number in each patterns represents the tasks assignment order of the node. Since the participating nodes in each pipeline need to be different, the same shapes do not appear in the same column.

Design example: Taking the $(5,3)$RS code in Fig. 2 as an example, we show the process of Algorithm 2. First, FullRepair obtains the maximum pipelined repair throughput via algorithm 1, which is 900 Mbps.

(1)*Own task assignment* in the example:

Algorithm 2 will first divide the parallel pipelining tasks according to t_{max} and assign the own task of each node.

First, Algorithm 2 sorts the nodes according to the available downlink bandwidths in descending order. As each node can be allocated from the total throughput t_{max} with the own task up to the speed of $\frac{Downlink_i}{k-1}$, the sorting order and the allocated bandwidth of each node's own task are: N_3, 500Mbps; N_2, 150Mbps; N_4, 150 Mbps; N_5,100 Mbps. Since N_5 is the last node to participate in the allocation, it only gets the remaining 100 Mbps of t_{max} as its own task.

Second, Algorithm 2 sorts nodes in descending order based on the the remaining available uplink bandwidth after their own tasks' allocation. Since the second ordering also numbers the order of pipelining tasks, the tasks' number and the bandwidths of own tasks of each node are: N_5, Task1, 100Mbps; N_2, Task2,150 Mbps; N_4, Task3,150 Mbps; N_3,Task4, 500 Mbps. The initial state before task assignments is shown in Fig. 3(a), as the allocation of each task are divided into k parts(including own tasks and sending tasks).

(2)*Sending task assignment* in the example:

The specific assignment process for each node is as follows:

- *Assignment of N_5 sending tasks* (Fig. 3(b)): First, N_5 will be assigned its own task Task1. After the task occupies 100 Mbps of its uplink bandwidth, the node has 500 Mbps left. Then, N_5 selects the top-ranked task with more remaining tasks in $T_{assigned}$ and $T_{unassigned}$. The strategy is shown in the figure as: we first search the blank space in the shaded column from the left to the right in the top unfilled row, and then search the the blank space in unshaded column from the right to the left row by row. For N_5, there is no blank space in the top row with a shaded column, so it will fill the blank space from the right to the left, and thus the last Task4 is allocated and the left 500 Mbps bandwidth is allocated.
- *Assignment of N_2 sending tasks* (Fig. 3(c)): Similar to the allocation process of N_5 node, after allocating its own task Task2, there is still some unused available uplink bandwidth. Since there is no shaded column in the top empty row, Task3 is allocated to N_2 and the first row is all allocated. At this point, N_2 still has 400 Mbps available uplink bandwidth that can be allocated, so in the collection of shadowed columns, Task1 is selected in the second row from the left to the right. When there is still 200Mbps left, the algorithm finds the blank space in unshaded columns of the second row from right to left: Task4, and Only 200Mbps of Task 4 is allocated as the speed of Task4 is 500Mbps.
- *Assignment of N_4 sending tasks* (Fig. 3(d)): After assigning the own task Task3 to N_4 , Task2, which has a blank space in the shaded column in the second row, is also assigned to N_4. The left available uplink bandwidth of N_4 and the left half of Task4 are both 300Mbps, so half of the Task 4 is assigned to N_4.
- *Assignment of N_3 sending tasks* (Fig. 3(e)): Task4 is first assigned as the own task of node N_3, resulting in the filled column and the completion of the pipeline construction of Task4. However, as Task4 shared by N_2 and N_4 in the second row, Task4 is divided into the tasks of two pipelines and N_3 becomes the hub of two pipelines.
- *Result* (Fig. 3(f)): So far, Algorithm 2 completes the assignment of all tasks in the example. In the final process for the tasks, Task4 will also be divided into Task4a and Task4b to form a repair scheduling network which consist of five pipelines.

In conclusion, FullRepair takes t_{max} as the total throughput of the whole repair process to schedule all the pipeline tasks. All the pipelines can be constructed and the parallel repair can be implemented, as the size of the task segment for each pipeline task is also the repair speed of the pipeline. Table III shows the tasks, task segments and destination nodes assigned to each node. The bold tasks are the own tasks of each node.

Table III: Task assignment of each node after the pipeline repair task scheduling algorithm in the (5,3)RS example

Node	Task	Task segment	Destination Node
N2	Task1	0-100	N5
	Task2	**100-250**	**R**
	Task3	250-400	N4
	Task4a	400-600	N3
N3	Task1	0-100	N5
	Task2	100-250	N2
	Task3	250-400	N4
	Task4a	**400-600**	**R**
	Task4b	**600-900**	**R**
N4	Task2	100-250	N2
	Task3	**250-400**	**R**
	Task4b	600-900	N3
N5	**Task1**	**0-100**	**R**
	Task4a	400-600	N3
	Task4b	600-900	N3

C. Discussion

Impact of multi-pipelines: When the multiple pipelines have overlapping nodes, these nodes indeed consume more CPU and DRAM resources in return for less bandwidth resources; nevertheless, bandwidth resources often dominate the repair performance in erasure-coded clusters (RP [12], PPT [13], PivotRepair [9]), so FullRepair still maintains its merits.

Comparison with single-pipeline algorithms: Compared with RP constructed on chain-like path, FullRepair takes uplinks and downlinks bandwidth of multiple nodes into pipeline construction (Algorithm 1), and thus can adapt to heterogeneous network environments. Compared with PPT and PivotRepair, which are also adapted to heterogeneous network environments, FullRepair starts multiple pipelines to enable $n-1$ available nodes to participate in repair (Algorithm 2), which makes full use of available bandwidth resources with the algorithm complexity of $O(n^2)$.

V. Evaluation

A. Implementation

To understand the actual repair performance of FullRepair in the clustered storage system, we implement the FullRepair prototype system in C++ and Python with about 3000 SLoCs, based on Intel ISA-L. FullRepair runs in a cluster of a master node and data nodes, similar to the Master-Slave architecture in HDFS. The master node plays the role of controlling the task flow, knows the bandwidth information in the entire cluster network, and enables calculating and allocating tasks to each data node based on Algorithms 1 and 2. All data nodes store the erasure-coded chunks in the system, which receives and executes pipelined repair tasks assigned to them.

B. Setup

We conduct cloud experiments for chunk repairs under the network bandwidth conditions of three representative storage system workloads: TPC-DS, TPD-H, and SWIM. The experiment is carried out on the FullRepair prototype system with 15 nodes built and deployed on Amazon EC2. In the evaluation, four sets of representative and commonly used RS coding parameters were selected as the system settings. The specific parameters include: (6, 4) (typical RAID-6 settings), (9, 6) (QFS [17] erasure code encoding parameters), (12, 8) (encoding parameters of Baidu Atlas [23]) and (14, 10) (encoding parameters of Facebook [6]). The size of the repaired data chunks is set to 64 MiB [24], and the total bandwidth value of each node is set to 1000 Mbps.

In the cloud experiment, to ensure that the experimental results are close to the real storage system, we let each type of workload generate 6000 groups of bandwidth distributions that are continuous in time, and randomly select 100 groups of bandwidth distributions having congested nodes from each distribution to test the repair performance of FullRepair, RP, PPT and PivotRepair. The test mainly examines the three key metrics of a repair task: overall repair time, algorithm calculation time and data transfer time. We take the average of all results under all bandwidth conditions as a basis for comparison.

C. Experiments

Experiment 1. Overall repair time: In the test of the bandwidth of the three types of workloads, for any (n,k) parameters, FullRepair outperforms the other three algorithms in terms of overall single-chunk repair time, as shown in in Figure 4. Compared to RP, FullRepair outperforms in all cases; in the case of (n,k) = (9, 6) in Fig 4(b), the overall repair time of FullRepair is reduced by 45.4% compared to RP. Compared with PPT, due to its brute force search of its algorithm, the repair time increases significantly with the increase of n, and FullRepair reduces the overall repair time significantly; in Fig 4(c) with (n,k) = (14, 10), the reduction can be by up to 62.93%. Compared with PivotRepair, the overall repair time of FullRepair is reduced for all data sets and coding parameter; the reduction reaches 33.19% for TPC-DS with (n,k) = (14, 10).

Experiment 2. Algorithm's calculation time: The average calculation time of the four algorithms of RP, PPT, PivotRepair and FullRepair for single-chunk repair (that is, the calculation time for generating pipeline scheduling in the master node) is shown in Figure 5 respectively under the bandwidth conditions of three different workloads. Due to traversing all pipeline trees, PPT's algorithm runtime is much larger than the remaining three algorithms in any case. For RP, when (n,k) is small, the calculation time of the algorithm is similar to that of PivotRepair and FullRepair; when the number of nodes increases, the iterative algorithm used in RP needs to constantly try pipeline combinations, and the calculation time gradually increases. As shown in Fig 5(c) , when n changes from 6 to 14, the calculation time of RP increases from $17.75\mu s$ to $12.7ms$. For PivotRepair and FullRepair, the time complexities of these two algorithms are $O(nlogn)$ and $O(n^2)$ respectively, so the average calculation time does not increase significantly as the node size increases. Although FullRepair is slightly slower than PivotRepair, its impact on the overall repair time is limited. In Fig 5(a), when $n = 14$, FullRepair reaches its maximum average algorithm computation time of 46.2 μs, which is much smaller than the overall repair time (measured in seconds).

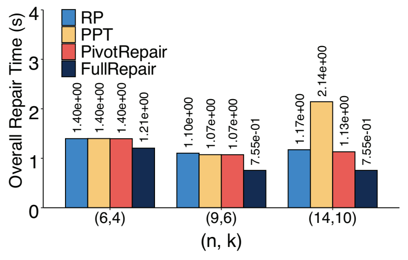
Experiment 3. Data transfer time: We show in Figure 6 the average data transmission time of the four algorithms of RP, PPT, PivotRepair and FullRepair for chunk repair under the bandwidth conditions of three different data sets (the time of data successfully repaired after the master node sends the pipeline task). It can be seen from the figure that RP has the longest data transmission time in all cases, PPT and PivotRepair have almost the same performance, and FullRepair takes the least time. This shows that when there are congested nodes in the network, the chain-like pipeline of RP has the weakest ability to bypass the congestion and reasonably utilize the uneven bandwidth of the network due to its low flexibility. Both PPT and PivotRepair use tree-like pipelines, which have a certain ability to bypass congested links, but still do not make full use of network bandwidth. FullRepair has the strongest ability to deal with uneven network conditions, and can make full use of all bandwidth resources in the network to speed up single-chunk repairs. FullRepair can reduce the data transmission time overhead of the existing parallel repair schemes RP, PPT and PivotRepair by up to 45.28% (Fig6(b), (n,k) = (9, 6)), 40.6% (Fig6(b), (n,k) = (9, 6)), 40.09% (Fig6(b), (n,k) = (9, 6)).

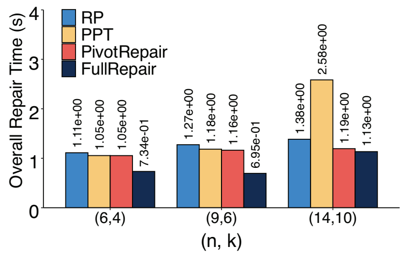
Experiment 4. Impact of slice size: We evaluate the single chunk repair time versus the slice size with a fixed bandwidth situation. We set the chunk size to 64 MiB and (n, k) is set to (6, 4). We vary the slice size ranging from 2 KiB to 1024 KiB. As can be seen from the Fig7, the chunk repair time of all methods decreases as the slice size increases, and FullRepair has less repair time than other methods for all slice sizes. the pipelined repair advantage of FullRepair is not affected by the slice size.

Experiment 5. Impact of chunk size: We evaluate the overall single chunk repair time versus chunk size for a fixed bandwidth. We set (n, k) to (6, 4). We vary the chunk size from 4 MiB to 64 MiB. As shown in the Fig8, as the chunk size increases, the repair time of all methods increases accordingly. In contrast, FullRepair, although the repair time increases linearly with chunk size, the repair time stays low and is significantly smaller than the other methods.

VI. Related Work

To address the data repair problem in erasure-coded systems, there are many repair strategies. Some strategies reduce the amount of data to be transmitted when reconstructing data by proposing various types of coding constructions [14], e.g., local repairable codes [2], [19], [25] that reduce the amount of data required for repair within a group, regenerating codes [26]–[28] that reduce the bandwidth required for data repair by utilizing the data encoding capacity of each node. There are also some strategies that do scheduling of the various phases, e.g., PPR [11], a fast single-chunk repair scheme via paralleling the repair operation as partial operations; FastPR [29], a fast predictive


(a) overall repair time in tpcds


(b) overall repair time in tpch

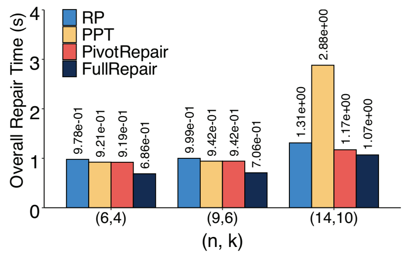
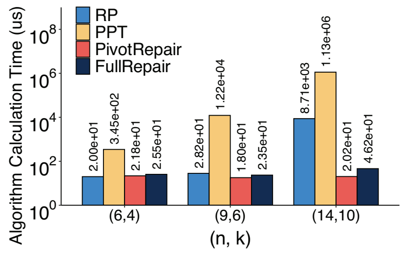
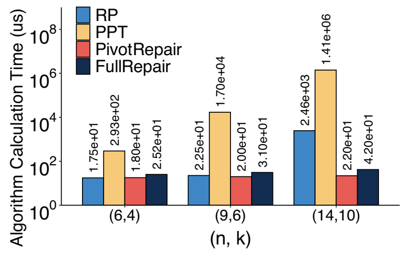

(c) overall repair time in swim

Figure 4: Experiment 1: Overall repair time in different (n,k)s.


(a) algorithm calculation time in tpcds


(b) algorithm calculation time in tpch

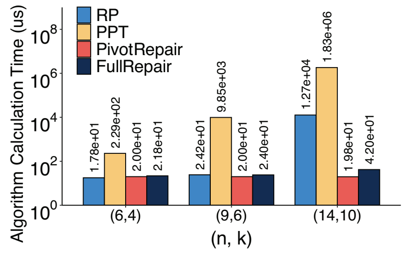
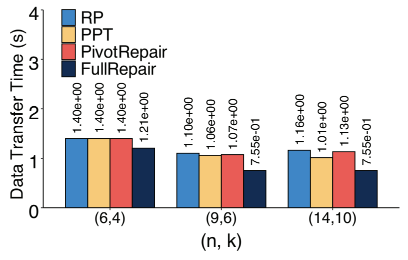
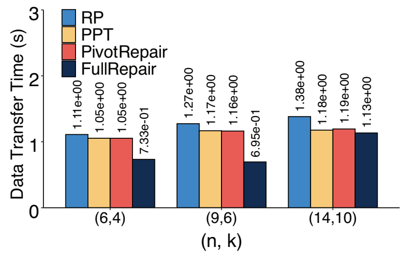

(c) algorithm calculation time in swim

Figure 5: Experiment 2: Algorithm calculation time in different (n,k)s.


(a) data transfer time in tpcds


(b) data transfer time in tpch

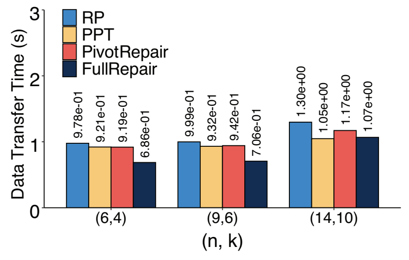

(c) data transfer time in swim

Figure 6: Experiment 3: Data transfer time in different (n,k)s.

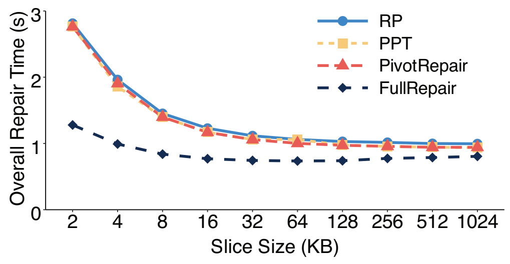


Figure 7: Experiment 4: Impact of slice size

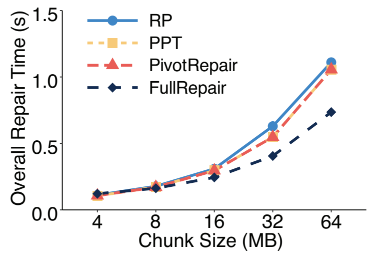


Figure 8: Experiment 5: Impact of chunk size

repair scheme; Dayu [30], a repair scheduling schemes based on free timeslots to cope with dynamic workload; RepairBoost [31], a fast full-node repair scheme via balancing and scheduling traffic carefully. FullRepair operates on RS codes that are widely deployed in production and aims to optimize the RS-based pipelined repair via scheduling the repair tasks.

Recent studies on erasure-coded repair strategies have begun to focus on re-allocating (rather than reducing) repair traffic to reduce repair time. This has led to pipelining techniques [32] for improving the efficiency of the repair task. RP [12] introduces repair pipelining into the field of erasure-coded repair by proposing chain-like pipelined repair, which significantly improves the performance of single-chunk recovery. Parallel pipeline tree (PPT) and parallel pipeline cross tree (PPCT) structures [13] can speed up the pipelined repair of single-chunk recovery and reduce repair time in a non-uniform traffic network environment. PivotRepair [9] further accelerates the tree pipeline construction algorithm to find the optimal tree pipeline at a very fast speed and solves the problem that the scheduling scheme of PPT is very time-consuming. FullRepair is a new pipelined repair strategy that operates on multiple pipelines instead of single pipelines that existing methods adopt, so as to achieve optimal erasure-coded repair by fully utilizing the network resources of all available nodes.

VII. Conclusions

In this paper, we propose a new pipelined repair strategy FullRepair for clustered storage systems, which is based on the idea of fully utilizing the available bandwidth of all non-failed nodes in the cluster by scheduling the repair tasks on multiple pipelines simultaneously, such that FullRepair can increase the network bandwidth utilization, thus improves the data repair performance. We analyze and calculate the maximum pipelined repair throughput of performing erasure-coded data repair for any given network condition, and propose the corresponding multi-pipeline scheduling greedy algorithm so that the scheduling of the pipelined repair task can achieve the optimality for any network bandwidth condition. Experiments demonstrate that FullRepair outperforms the existing state-of-the-art pipelined repair schemes RP, PPT, and PivotRepair in terms of single-chunk repair time.

Acknowledgment

This work was supported in part by the National Key Research and Development Program of China for Young Scholars (No. 2021YFB0301400), National Natural Science Foundation of China (No. 62272185), Key Research and Development Program of Hubei Province (No. 2021BAA189), National Natural Science Foundation of China (No. 61821003), and Key Laboratory of Information Storage System Ministry of Education of China. The corresponding author is yuchonghu@hust.edu.cn.

References

[1] H. Weatherspoon and J. D. Kubiatowicz, "Erasure Coding Vs. Replication: A Quantitative Comparison," in *Proc. of Springer IPDPS*, Mar 2002.

[2] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure Coding in Windows Azure Storage," in *Proc. of USENIX ATC*, Jun 2012.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proc. of USENIX OSDI*. USENIX Association, 2006, pp. 307–320.

[4] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in hdfs." in *Proc. of USENIX FAST*, 2015, pp. 213–226.

[5] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan, "Availability in Globally Distributed Storage Systems," in *Proc. of USENIX OSDI*, Oct 2010.

[6] K. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran, "A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster," in *Proc. of USENIX HotStorage*, 2013.

[7] Y. Li, R. Miao, H. H. Liu, Y. Zhuang, F. Feng, L. Tang, Z. Cao, M. Zhang, F. Kelly, M. Alizadeh *et al.*, "Hpcc: high precision congestion control," in *Proc. of ACM SIGCOMM*. ACM, 2019, pp. 44–58.

[8] H. Zhang, J. Zhang, W. Bai, K. Chen, and M. Chowdhury, "Resilient datacenter load balancing in the wild," in *Proc. of ACM SIGCOMM*. ACM, 2017, pp. 253–266.

[9] Q. Yao, Y. Hu, X. Tu, P. P. Lee, D. Feng, X. Zhu, X. Zhang, Z. Yao, and W. Wei, "Pivotrepair: Fast pipelined repair for erasure-coded hot storage," in *2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2022, pp. 614–624.

[10] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin, "Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage," in *Proc. of ACM SYSTOR*. ACM, 2014, pp. 1–7.

[11] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi, "Partial-parallel-repair (PPR): a distributed technique for repairing erasure coded storage," in *Proc. of ACM Eurosys*. ACM, 2016, p. 30.

[12] R. Li, X. Li, P. P. Lee, and Q. Huang, "Repair pipelining for erasure-coded storage," in *Proc. of USENIX ATC*, 2017, pp. 567–579.

[13] Y. Bai, Z. Xu, H. Wang, and D. Wang, "Fast recovery techniques for erasure-coded clusters in non-uniform traffic network," in *Proceedings of the 48th International Conference on Parallel Processing*, 2019, pp. 1–10.

[14] J. S. Plank, "Erasure codes for storage systems: A brief primer," *Login: The USENIX Magzine*, vol. 38, no. 6, pp. 44–50, 2013.

[15] I. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[16] "OpenStack Swift Object Storage Service," http://swift.openstack.org.

[17] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly, "The Quantcast File System," in *Proc. of ACM VLDB*, 2013.

[18] D. S. Papailiopoulos, J. Luo, A. G. Dimakis, C. Huang, and J. Li, "Simple regenerating codes: Network coding for cloud storage," in *Proc. IEEE INFOCOM*. IEEE, 2012, pp. 2801–2805.

[19] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing Elephants: Novel Erasure Codes for Big Data," in *Proc. of ACM VLDB Endowment*, 2013, pp. 325–336.

[20] "TPC-DS," http://www.tpc.org/tpcds/.

[21] "TPC-H," http://www.tpc.org/tpch/.

[22] "SWIM," https://github.com/SWIMProjectUCB/SWIM.

[23] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong, "Atlas: Baidu's key-value storage system for cloud data," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[24] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.

[25] D. S. Papailiopoulos and A. G. Dimakis, "Locally repairable codes," *IEEE Trans. on Information Theory (TIT)*, 2014.

[26] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran, "Network Coding for Distributed Storage Systems," *IEEE Trans. on Information Theory (TIT)*, vol. 56, no. 9, pp. 4539–4551, Sep 2010.

[27] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Guyot, E. E. Gad, and Z. Bandic, "Opening the Chrysalis: On the Real Repair Performance of MSR Codes." in *Proc. of USENIX FAST*, 2016, pp. 81–94.

[28] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy *et al.*, "Clay codes: moulding MDS codes to yield an MSR code," in *Proc. of USENIX FAST*, 2018, p. 139.

[29] Z. Shen, X. Li, and P. P. Lee, "Fast predictive repair in erasure-coded storage," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2019, pp. 556–567.

[30] Z. Wang, G. Zhang, Y. Wang, Q. Yang, and J. Zhu, "Dayu: Fast and low-interference data recovery in very-large storage systems." in *USENIX Annual Technical Conference*, 2019, pp. 993–1008.

[31] S. Lin, G. Gong, Z. Shen, P. P. Lee, and J. Shu, "Boosting full-node repair in erasure-coded storage." in *USENIX Annual Technical Conference*, 2021, pp. 641–655.

[32] C. V. Ramamoorthy and H. F. Li, "Pipeline architecture," *ACM Computing Surveys (CSUR)*, vol. 9, no. 1, pp. 61–102, 1977.