PivotRepair: Fast Pipelined Repair for Erasure-Coded Hot Storage

Qiaori Yao†, Yuchong Hu†, Xinyuan Tu†, Patrick P. C. Lee‡, Dan Feng†,
Xia Zhu*, Xiaoyang Zhang*, Zhen Yao*, Wenjia Wei*
†*Huazhong University of Science & Technology* ‡*The Chinese University of Hong Kong* **HUAWEI*

***Abstract*—Erasure coding is commonly used as a storage-efficient redundancy method for fault tolerance in cold storage. Recent studies have begun to explore the use of erasure coding in hot storage, which requires fast online recovery to preserve read performance. However, existing erasure-coded repair strategies cannot effectively handle frequent and rapidly-changing network congestions in hot storage clusters. In this paper, we present the notion of *pivots*, which refer to the storage nodes with sufficient available downlink and uplink bandwidths in a congested hot storage network. We propose PivotRepair, a pivot-based pipelined single-chunk repair technique that leverages pivots for enabling the fast construction of a pipelined repair tree that bypasses congested links. We further propose an adaptive scheduling strategy to improve full-node repair performance. We prototype PivotRepair and show that the repair time of a single-chunk repair and a full-node repair can be reduced by up to 71.27% and 16.50%, respectively, over state-of-the-art repair schemes.**

I. Introduction

To maintain data reliability at low cost, today's distributed storage systems adopt erasure coding to protect data with a low degree of redundancy [24], [34], while preserving the same fault tolerance as replication [52]. For instance, Azure [24] and Facebook [34] adopt erasure coding to reduce redundancy to 1.33× and 1.4×, respectively, instead of 3× in three-way replication [14], [21]. An erasure code works by encoding k uncoded fixed-size units, called *chunks*, into n ($n > k$) coded chunks, such that any k out of n coded chunks can rebuild the k uncoded chunks. By distributing n coded chunks across n storage nodes, a distributed storage system can provide fault tolerance against node failures in clustered [20], [24], [42] or geo-distributed storage [11], [17], [34], [46].

Although erasure coding improves storage efficiency, it results in a high repair cost. Repair is triggered when reading unavailable data caused by transient failures or reconstructing lost data from permanent failures. In either case, a *requestor* (i.e., a node where the lost chunk is reconstructed) needs to read multiple available chunks from multiple *helpers* (i.e., the surviving nodes that store the available chunks) for reconstruction, thereby leading to substantial *repair traffic* (i.e., the amount of data transferred for repair). Prior studies propose repair-friendly erasure codes that reduce the repair traffic (e.g., regenerating codes [19], [36], [41], [50] and locally repairable codes [24], [37], [47]). Recently, new repair strategies focus on distributing (instead of reducing) the repair traffic to reduce the repair time. For example, PPR [33] parallelizes the repair via multiple partial operations to distribute the repair traffic across helpers; RP [28] pipelines the repair across helpers in sub-blocks (called slices) to evenly distribute repair traffic; PPT [12] pipelines the repair in a tree-like structure in non-uniform traffic environments.

While erasure coding is popularly used in cold storage, recent studies explore erasure coding for serving hot data [15], [40], [54], and modern data centers also perform erasure coding on hot storage nodes [24]. Unlike in cold storage, hot storage clusters host frequently accessed data, and hence their services often require low I/O latencies. Once any data chunk becomes unavailable due to transient or permanent failures, an immediate and fast repair operation is critical to reconstruct any unavailable data chunk and maintain data availability.

Achieving immediate and fast repair for erasure-coded hot storage is challenging, as the *available bandwidth* for repair jobs is often limited. For example, practical storage systems often rate-throttle the available bandwidth for repair jobs [24], [48]. Also, the network bandwidth is often shared by both repair and foreground jobs, and application workloads may periodically transfer a large volume of data that causes network congestion to repair jobs [31]. Our measurement analysis in §III-A on three typical hot data workloads (namely TPC-DS [9], TPC-H [10], and SWIM [16]) shows that the available bandwidth highly fluctuates, and different nodes may experience congestions at different times. State-of-the-art pipelined repair strategies either cannot rapidly bypass network congestion (e.g., RP [28]) or cannot solve for the most suitable pipelined tree in a short time (e.g., PPT [12]).

Nevertheless, our measurement analysis (§III-A) also shows that while there exist congested nodes in a hot storage cluster, there also likely exist uncongested nodes that have sufficient available downlink and uplink bandwidths. Thus, our main idea is to exploit such uncongested nodes (called *pivots*) to construct a pipelined tree (composed of multiple leaf-to-root paths) for repairing unavailable chunks, such that the pipelined tree can (i) effectively bypass congestion by making the pivots relay the repair traffic and (ii) be quickly initialized and constructed via the pivots (§IV).

In this paper, we present PivotRepair, a pivot-based repair technique that aims for fast pipelined repair in erasure-coded hot storage. Our contributions include:

- We conduct measurement analysis and show that in hot storage clusters, congestion is frequent and rapidly changing, while some nodes (i.e., pivots) still have abundant bandwidth.
- We design an $O(n \log n)$ greedy algorithm for PivotRepair (recall that n is the number of coded chunks) that exploits

This work was supported in part by the National Key Research and Development Program of China for Young Scholars (No. 2021YFB0301400) and National Natural Science Foundation of China (No. 61872414), Key Laboratory of Information Storage System Ministry of Education of China, and Research Grants Council of HKSAR (AoE/P-404/18). The corresponding author is Yuchong Hu (yuchonghu@hust.edu.cn).

pivots to generate a pipelined tree. We prove that our algorithm is optimal, in that it maximizes the bottlenecked bandwidth. We further propose an adaptive scheduling strategy to enhance full-node repair performance.
- We prototype and evaluate PivotRepair on Amazon EC2. Compared to RP [28] and PPT [12], PivotRepair reduces the repair time for a single-chunk repair and a full-node repair by up to 71.27% and 16.50%, respectively. Our prototype is open-sourced at: https://github.com/YuchongHu/PivotRepair.

II. Background

A. Basics of Erasure Coding

The literature (see survey [39] and §VI) has various proposals on erasure coding constructions, among which Reed-Solomon (RS) codes [45] are popularly adopted in production (e.g., HDFS [42], Ceph [53], Swift [5], and QFS [35]). An RS code is associated with two parameters (n,k), where $k<n$, and is applied to a set of chunks of fixed size (e.g., 64 MiB [21]). An (n,k) RS code encodes k *data chunks* into $n-k$ equal-size *parity chunks*, such that any k out of the n data/parity chunks (collectively called a *stripe*) suffice to rebuild the original k data chunks. Specifically, for a stripe composed of data and parity chunks denoted by D_i $(1 \le i \le k)$ and P_j $(1 \le j \le n-k)$ respectively, the parity chunks are calculated from a linear combination of the data chunks as $P_j = \sum_{i=1}^{k} \alpha_j^{i-1} D_i$, where α_j^{i-1} $(1 \le i \le k$ and $1 \le j \le n-k)$ are encoding coefficients constructed from the Vandermonde matrix [13]. For example, for a $(5,3)$ RS code, its first parity chunk $P_1 = D_1 + \alpha_1 D_2 + \alpha_1^2 D_3$. Additions and multiplications are based on Galois Field GF(2^w) [45] over w-bit words, such that the words at the same offset of k data chunks are encoded to generate the corresponding words in the parity chunks [28]. In particular, the addition of two chunks is done by bitwise-XOR operations, and multiplying a chunk by a constant is operated via multiplying each word of the chunk by the constant.

B. Linearity

Erasure codes are in essence based on linear encoding, so the repair operations can be performed via a linear addition. For example, for a $(5,3)$ RS code, we can repair the first data chunk $D_1 = P_1 + \alpha_1 D_2 + \alpha_1^2 D_3$. The linearity of repair operations implies two properties [33]:
- **Property 1: Additions keep the data size unchanged.** The XOR-based additions ensure that the addition results have the same size as the original chunks. For example, to repair D_1, all partial addition results (i.e., P_1, $P_1 + \alpha_1 D_2$, and $P_1 + \alpha_1 D_2 + \alpha_1^2 D_3$) have the same size as that of D_1.
- **Property 2: Additions are associative.** The order of linear additions does not alter the results. For example, both $(P_1 + \alpha_1 D_2) + \alpha_1^2 D_3$ and $P_1 + (\alpha_1 D_2 + \alpha_1^2 D_3)$ can decode D_1.

Property 1 simplifies the parallelization and pipelining of repair operations since all the additions of a repair operation always handle fixed-size chunks, while Property 2 enables a repair operation to flexibly perform additions in any order. Recent efficient erasure-coded repair schemes [12], [28] (see §II-C for details), as well as our proposed method PivotRepair, also build on these two properties.

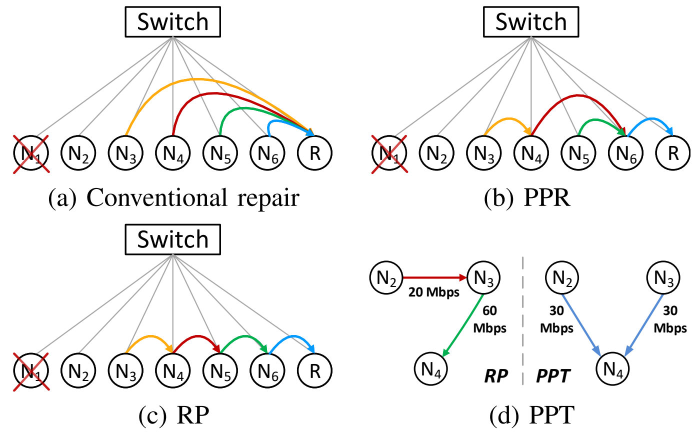


Figure 1: Examples of conventional repair, PPR, RP, and PPT. Note that in all repair methods, any transmission between two nodes should go through the switch as in common storage clusters, but for brevity, we do not always show the switch in the figures of this paper (e.g., Figure 1(d)).

C. Repair

Erasure coding has the well-known *repair problem* [24], [38], [42], [47], due to its excessive bandwidth usage when repairing any unavailable data. Figure 1(a) depicts the congestion issue when repairing a failed chunk with a $(6,4)$ RS code (we call this *conventional repair*). Specifically, for an (n,k) RS code, the repair of a failed chunk (say, stored in N_1) requires the requestor (denoted by R) to download k of the remaining chunks of the same stripe from k different available helpers; this leads to the congestion at the requestor and increases the overall repair time. For example, in Figure 1(a), the requestor R downloads one chunk from each of the helpers (say, N_3, N_4, N_5, N_6), so the downlink of the requestor is four times more congested than each of the helpers. To reduce repair traffic, many repair-friendly erasure codes (e.g., [19], [24], [25], [36], [41], [43], [47]) have been proposed, but they all do not address network congestion during the repair process as they assume that the requestor downloads the data for reconstruction (albeit with less repair traffic). To address congestion in erasure-coded repair, three recent repair approaches are proposed as follows.
- **Partial-Parallel-Repair (PPR) [33]** (Figure 1(b)): PPR decomposes a repair operation into parallel partial sub-operations that are performed simultaneously in multiple helpers. It improves repair performance by distributing the repair traffic more evenly across the network links.
- **Repair Pipelining (RP) [28]** (Figure 1(c)): RP [28] observes that PPR does not fully balance the distribution of repair traffic (e.g., N_6 in Figure 1(b) has the most repair traffic), so the most congested helper still bottlenecks the repair performance. Thus, RP arranges all helpers as a chain-like path, and pipelines the repair operation across helpers in sub-chunks (called slices), such that no link transmits more traffic than others (i.e., no bottlenecked links).

- **Parallel Pipeline Tree (PPT) [12]** (Figure 1(d)): PPT [12] finds that a chain-like path (like RP) may bottleneck the repair performance by the slowest link of the path (e.g., $N_2 \rightarrow N_3$ with an available bandwidth of 20 Mb/s in Figure 1(d)). Alternatively, PPT's pipelined tree replaces this link with another two links with the same receiver, such that even if the bandwidth of each link has been reduced by half due to the same receiver, the slowest link still has a higher available bandwidth (e.g., 30 Mb/s in Figure 1(d)) than RP. However, PPT needs to search all link bandwidths via permutation enumeration to maximize the slowest link bandwidth, thereby incurring an exponential time complexity (based on Bell number B_k [12] for an (n,k) RS code).

Compared to PPR, both RP and PPT introduce the pipelining technique for improved repair performance. Thus, in this paper, we also focus on the pipelined repair. In general, the throughput of a pipeline cannot be better than that of its slowest stage, so *our main goal of the pipelined repair is to maximize the bandwidth of its slowest stage*.

III. Observations and Motivation

Addressing the congestion issue in the repair problem of erasure coding is challenging in hot storage, since the bandwidth is often shared by both application and repair jobs. Thus, we first study the bandwidth details of hot storage workloads and identify two observations (§III-A). Based on the observations, we argue that in hot storage, state-of-the-art pipelined repair strategies (i.e., RP and PPT) cannot efficiently perform the repair (§III-B).

A. Measurement Analysis

Hot storage workloads require fast response time, such as in quick decision making [6] and web content [4]. Thus, we conduct measurement analysis on three hot storage workloads to motivate our study, namely: (i) TPC-DS [9], which is a popular decision support benchmark featuring one throughput metric of queries; (ii) TPC-H [10], which is a classical decision support benchmark featuring business databases; and (iii) SWIM [7], which is a MapReduce trace on a 3000-machine cluster at Facebook within 1.5 months. To evaluate the workloads, we set up a Hadoop cluster of 16 machines with the edge bandwidth of 1 Gbps (configured by the Linux `tc` command [8]). For TPC-DS and TPC-H, we generate traces of size 100 GB atop Hive; for SWIM, we generate a scaled-down trace on 16 machines atop MapReduce. By evaluating the three traces in our cluster, we make two observations about network congestion that motivate the design of PivotRepair.

Our measurement analysis focuses on the *used node bandwidth* of each node, defined as the larger value of the used downlink and uplink bandwidths of the node incurred by applications. This indicates the congestion level of the node. Correspondingly, the *available node bandwidth* is defined as the remaining node bandwidth for the repair job, which is the smaller value of the available downlink and uplink bandwidths of the node. Here, we measure the link bandwidth using the Linux `nload` command [3] to monitor network traffic and bandwidth usage in real time.

Usage rate	Traces		
	TPC-DS	TPC-H	SWIM
>90%	37.1%	57.8%	23.6%
>95%	37.6%	61.2%	24.4%
=100%	40.2%	67.3%	29.7%

Table I: Percentage of the total time for congested nodes with $C_v > 0.5$.

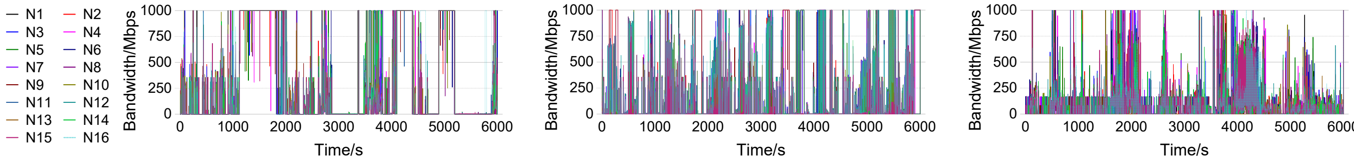
Observation 1: Figure 2 shows the used node bandwidth distribution over 16 nodes within 6000 s (measured at one-second intervals). We observe each of 16 nodes experiences congestion (i.e., the used node bandwidth is close to 1 Gbps) at different times, and the set of congested nodes at each one-second interval varies frequently and rapidly. Also, it is shown that even a single congested link may significantly degrade the jobs that have communicating tasks traversing the link [18]. Thus, it is likely that *the performance of a repair job is severely bottlenecked by frequent and rapidly-changing congested nodes caused by application jobs in hot storage*.

Observation 2: Table I shows the heterogeneity of the used node bandwidth across different application workloads when congestion happens. To measure the congestion, we use the *usage rates* [27] (i.e., the percentage of node bandwidth usage) that range from 90% to 100% to indicate the presence of congestion. We also use the coefficient of variation C_v (i.e., the ratio of the standard deviation to the mean) of the average used node bandwidth over the 16 nodes in each one-second interval (as in [18]), so as to show the extent of bandwidth heterogeneity (e.g., $C_v = 0$ means all the nodes use identical bandwidth). We find that the nodes with ratio $C_v > 0.5$ account for up to 67.3% of the total time under congestion, meaning that when congestion happens, the used node bandwidths of different nodes are heterogeneous. In other words, the available node bandwidths for repair are also heterogeneous (assuming that each node has 1 Gbps edge bandwidth for all application and repair jobs). Thus, it is likely that *during repair, even if some nodes are congested, there still exist some uncongested nodes with relatively sufficient available downlink and uplink bandwidths.*

B. Motivation

Observation 1 shows that state-of-the-art pipelined repair strategies (i.e., RP [28] and PPT [12]) cannot efficiently cope with hot storage, specified as follows.

First, RP fails to handle the frequently congested nodes during repair, as RP runs all the repair stages in series along a chain-like pipelined path where each helper transfers the same amount of data. Take Figure 3 for example, and consider the same setting as in Figure 1 with heterogeneous available downlink and uplink bandwidths of each node for repair. Figure 3(a) shows that RP requires each helper to transfer data from its predecessor to its successor, so the most congested node N_5 (which has the downlink bandwidth of 200 Mbps) will bottleneck RP.



(a) TPC-DS (b) TPC-H (c) SWIM

Figure 2: Used Node Bandwidth Distribution by application workloads in hot storage.

Second, PPT cannot adapt to the rapidly-changing congested nodes, since it takes a long time to construct the pipelined tree and cannot quickly adapt to the available bandwidths on-the-fly based on the real-time link states. Figure 3(b) shows that PPT needs to enumerate all possible pipelined trees and examine their slowest link, and the enumeration time is based on the Bell number B_k [12], which increases exponentially with k. Thus, it is challenging to conduct enumeration quickly for rapidly-changing congestion for a large k. For example, in Figures 7(d)-(f), when $k=4$, the enumeration time of a single-chunk repair is only hundreds of milliseconds. However, when $k=10$, the enumeration time rises up to thousands of seconds, which may not be practical for a single-chunk repair with $(14,10)$ RS codes deployed in Facebook [43], [47].

While RP and PPT fail to handle congested nodes efficiently in hot storage, Observation 2 shows the existence of uncongested nodes (e.g., N_4 in Figure 3(c), with both sufficient available downlink and uplink bandwidths), which motivates us to leverage them to improve the pipelined repairs.

Our main idea is (i) to bypass the congested nodes (outperforms RP) using the uncongested nodes to relay the repair traffic, and (ii) to accelerate the pipelined tree construction (outperforms PPT) using the uncongested nodes to construct the tree in advance, which will be specified in §IV-A.

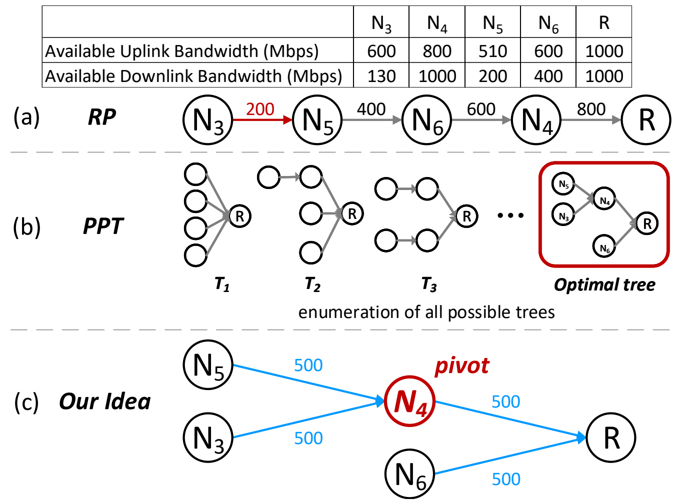


Figure 3: Motivating example. Here, we assume that the available bandwidth between two nodes (e.g., from N_3 to N_5 in Figure 3) is the smaller one of N_3's upload bandwidth and N_5's download bandwidth.

IV. PivotRepair

A. Overview

We propose a notion of *pivots* to indicate the uncongested nodes in a storage network. Our goal is to design a pivot-based pipelined repair technique, namely PivotRepair, to construct an optimal and fast-constructed pipelined repair tree. By "optimal", we mean that the tree bypasses as many congested nodes as possible and has the maximum bandwidth of its slowest stage.

To this end, PivotRepair uses pivots to form a pipelined tree, specified as follows. First, PivotRepair lets the requestor be the root node of the tree. Then it selects k helpers from $n-1$ surviving nodes. Among the k helpers, it selects the uncongested helpers as pivots, which serve as the non-leaf nodes of the tree. The pivot-based non-leaf nodes can be used to relay the repair traffic to avoid congestion, and also can be used to determine parts of the pipelined tree quickly.

We use Figure 3 as an example to show the benefits of PivotRepair. First, we can let the pivot N_4 be a non-leaf node of the tree, while the congested nodes (i.e., N_3, N_5, and N_6) only serve as the leaf nodes. N_4 can relay the data from N_3, N_5, and N_6 to fully utilize its sufficient downlink and uplink bandwidths, while bypassing the congested downlinks of N_3, N_5, and N_6. Note that N_4, which has 1,000 Mbps downlink bandwidth, can relay the data from N_3 and N_5 by providing two downlinks with 500 Mbps bandwidth each. This is in contrast to RP, which can be bottlenecked by congested nodes. Second, the pivot N_4 can be quickly selected by checking each node's available downlink and uplink bandwidths, so the non-leaf nodes of the pipelined tree can be quickly determined. In this way, the whole pipelined tree can also be quickly constructed. This is more suitable to cope with the rapidly-changing congested nodes than PPT, which requires enumerating all possible trees (around $k!$ trees [12]) and is time-consuming (e.g., Figure 3(b) has to check 24 trees). As shown in Figure 3(c), we let N_4 become a child of the requestor and also become a non-leaf node (that has at least one child). For the remaining nodes N_3, N_5, and N_6 (which will serve as leaf nodes), we only need to check at most seven possible trees to find the optimal tree (i.e., each of N_3, N_5, and N_6 is a child node of either N_4 or R, while N_4 has at least one child node).

B. Algorithm

We define the bandwidth of the slowest link of the repair pipelined tree as *minimum bandwidth* of the tree (denoted by B_{min}); PivotRepair aims to maximize B_{min}. We then design PivotRepair based on two major steps: *inserting* and *replacing*. Specifically, PivotRepair constructs the tree by (i) inserting

Algorithm 1 Tree Construction

Input: nodes bandwidths, requestor R
Output: an optimal pipelined repair tree T

1: **procedure** MAIN
2: $T = R$
3: $\mathcal{S}$ = set of sorted k pivots descended by $theo(\cdot)$
4: INSERTING
5: REPLACING
6: **return** T
7: **end procedure**
8: **function** INSERTING
9: Q is empty // Q is a priority queue based on $prac(\cdot)$
10: Q.push(R)
11: **for** each node $N_i \in \mathcal{S}(1 \leq i \leq k)$ **do**
12: N_j = Q.pop()
13: $T.N_j \rightarrow$ new_child $= N_i$
14: Q.push(N_i)
15: Q.push(N_j)
16: **end for**
17: **end function**
18: **function** REPLACING
19: L = set of leaf nodes in T
20: l = number of leaf nodes in T
21: L' = $L \cup \{the\ unselected\ nodes\}$
22: L^* = set of top l nodes in L' with largest $up(\cdot)$
23: $L_{replaced}$ = $L - L^*$
24: **for** each node $N_i \in L^*(1 \leq i \leq k)$ **do**
25: **if** $N_i \notin L$ **then**
26: Select one of the remaining nodes of $L_{replaced}$ as N_j
27: Replace N_j in T with N_i
28: **end if**
29: **end for**
30: **end function**

k pivots (sorted by node available bandwidth) one by one to construct a preliminary tree that aims to maximize B_{min}, and (ii) replacing some leaf nodes with those nodes that are not selected in the inserting step but have higher available uplink bandwidths. In this way, the inserting step ensures that all non-leaf nodes are constructed by pivots to bypass congested non-leaf nodes, so that the pipelined tree has available link bandwidth between non-leaf nodes. The replacing step further increases the available bandwidth of the links connected to leaf nodes to bypass congested leaf nodes.

We define a set of notations as follows. We denote available uplink and downlink bandwidths of node N_i by $up(i)$ and $down(i)$, respectively. In theory, each available node bandwidth (denoted by $theo(i)$) is $min\{up(i), down(i)\}$, while in practice, the node's downlink bandwidth is shared by its multiple child nodes. Thus, we denote its practical available node bandwidth by $prac(i)$ which is $min\{up(i), avgDown(i)\}$. Here, $avgDown(i) = down(i)/c$ is the average available bandwidth for each downlink of N_i connected to its c child nodes.

Algorithm 1 details the inserting and replacing steps:

Preparation: Before inserting, the requestor serves as the root node of the tree (Line 2). Then it sorts the $n-1$ surviving nodes in descending order of $theo(\cdot)$, and obtains the set of sorted k pivots that have the largest $theo(\cdot)$, denoted by S.

Step 1 (Inserting): We first create a priority queue Q based on $prac(\cdot)$ (Line 9). Q is initialized with the requestor as the first element (Line 10). For each of k pivots N_i (Line 11), Q is to help determine the inserting place of each pivot by choosing a node N_j out of the current Q to serve as the pivot's parent node, such that this chosen node N_j has the largest $prac(\cdot)$ (Line 12), and the preliminary tree T inserts N_i as its node N_j's child node (Line 13). Then we update Q via adding N_i and its parent node N_j (Lines 14-15). Finally, we can obtain the preliminary tree T after inserting the k pivots.

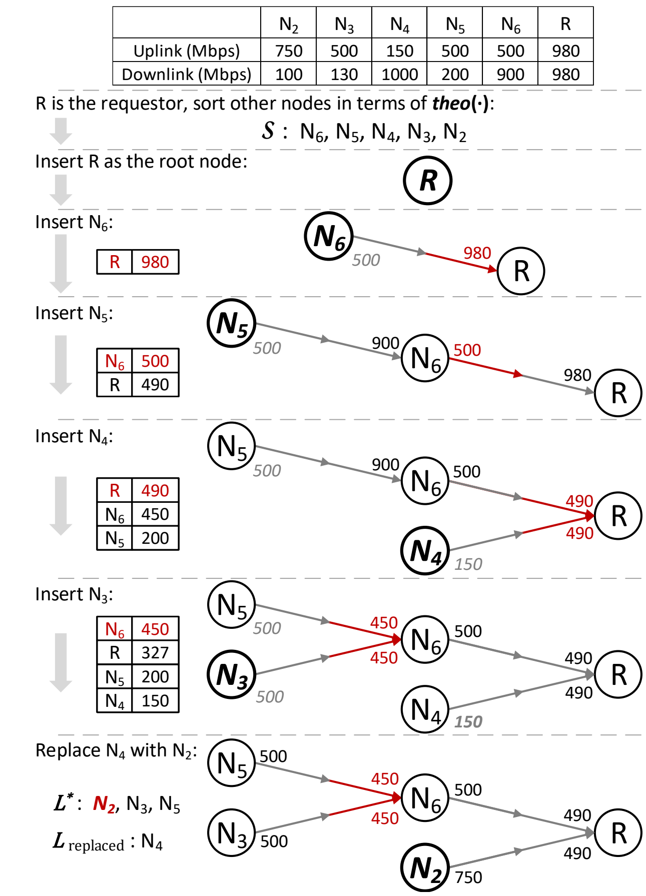


Figure 4: Illustration of Algorithm 1. Suppose that N_1 fails and $N_2, N_3, \ldots, N_6$ are helpers. The table in each Inserting step represents the current priority queue Q (including the nodes and their $prac(\cdot)$ values) before each pivot is to be inserted.

Step 2 (Replacing): To replace the preliminary tree's leaf nodes with those nodes that are not selected in inserting but have higher available uplink bandwidths, we first record the set of all l leaf nodes (denoted by L) of the preliminary tree after Step 1 (Inserting) (Lines 19-20). Let L' be the union of L and the set of those nodes that are not selected during inserting (Line 21). We then sort all nodes in L', and obtain the set of l nodes that has the largest uplink bandwidths, denoted by L^* (Line 22). Next, we find the preliminary tree's leaf nodes that do not belong to L^* (i.e., they have lower uplink bandwidths than those in L^*), denoted by $L_{replaced}$ (Line 23), and replace them with those nodes that are not selected in T but in L^* (Lines 24-29).

Figure 4 illustrates Algorithm 1 with $(n,k) = (6,4)$. PivotRepair first inserts the requestor into the tree and then sorts the $n-1=5$ helpers in terms of $theo(\cdot)$. PivotRepair finds $k=4$ pivots $S=\{N_6,N_5,N_4,N_3\}$ based on $theo(\cdot)$ in descending order, and then inserts the pivots one by one while satisfying that each pivot's parent node has the largest $prac(\cdot)$ for each insert to increase B_{min}. Lastly, at the replacing step, the leaf-node N_4 is replaced by N_2 to increase B_{min}.

Note that Algorithm 1 only ensures that each inserted pivot of the tree can achieve its current maximum B_{min} (i.e., local optimum), but it is not sure that Algorithm 1 can obtain the tree that has the final maximum B_{min} (i.e., global optimum). Thus, we will prove Algorithm 1's optimality in §IV-C.

C. Optimality and Complexity

Algorithm 1 aims to maximize the minimum bandwidth and accelerate the pipelined tree construction, and next we show its optimality in bandwidth (Theorem 1) and low time complexity.

Lemma 1. *For any pipelined tree T,*

$$B_{min}=min\{min\{S_{nl}\},min\{S_l\}\},$$

where S_{nl} is the set of $prac(\cdot)$ of the non-leaf nodes, and S_l is the set of $up(\cdot)$ of the leaf nodes.

Proof: For any tree T, its B_{min} is limited in three cases: (a) *up* in non-leaf nodes; (b) *avgDown* in non-leaf nodes; (c) *up* in leaf nodes. We define the set of bandwidths in the above cases as S_a, S_b, and S_c respectively. Clearly, we have

$$B_{min}=min\{min\{S_a\},min\{S_b\},min\{S_c\}\}, \quad (1)$$

$$S_{nl}=S_a\cup S_b, S_l=S_c \quad (2)$$

Based on Equations (1) and (2), we can have

$$B_{min}=min\{min\{S_{nl}\},min\{S_l\}\}. \quad \square$$

Lemma 2. *For any pipelined tree T constructed by the Inserting step, it achieves optimal $min\{S_{nl}\}$.*

Proof: We prove via mathematical induction, in which we have each step established by finding the contradiction that no other tree can achieve any higher $min\{S_{nl}\}$, and thus the tree T has the optimal $min\{S_{nl}\}$. The details are shown in Appendix. ☐

Lemma 3. *For any pipelined tree T constructed by the Inserting step, the Replacing step can change T into a pipelined tree with optimal B_{min}.*

Proof: We prove by cases on the bandwidths of the tree's leaf nodes, and find that the replacing step always makes the tree have optimal B_{min}. The details are shown in Appendix. ☐

Theorem 1. *For any pipelined tree T constructed by Algorithm 1, it achieves optimal B_{min}.*

Proof: The theorem holds based on Lemmas 2 and 3. ☐

The overall time complexity of Algorithm 1 is $O(n\log n)$. Specifically, all sorting operations in both preparation and replacing take $O(n\log n)$ time, while inserting only needs $O(n\log n)$ time to finish. The main reason is that we leverage a priority queue to rapidly choose the inserting place with the largest $prac(\cdot)$ for each pivot. Note that for each pivot, it is the priority queue that allows us to choose an optimal inserting place in $O(\log n)$ time, without traversing all the tree's existing nodes in $O(n)$ time. Experiment 2 in §V-C shows Algorithm 1's low running time even with a large n.

D. Slice-level Repair

Although Algorithm 1 has optimal (maximized) minimum bandwidth and low time complexity, repairing a single chunk with a large size (e.g., 64 MiB [21]) may still take a too long time to match the dynamics of network links during the repair process. It means that if we repair chunks one at a time, it will delay the total time to repair under the rapidly changing link-state (which may be serving short requests) in hot storage.

Thus, PivotRepair decomposes a single chunk into multiple slices [28]. For RS codes, a slice can be as small as one byte (§II-A), which is much smaller than the chunk size, so a single slice can be quickly transferred. When multiple slices are being transferred in parallel in the tree, the helpers constitute a pipeline of transferring slices, such that the pipelined tree can exploit all their bandwidth resources rather than a single one, while hiding the computation overhead at the same time. In this way, PivotRepair can fit in the rapidly changing link state in hot storage by finishing repair instantly.

E. Enhancing Full-Node Repair with Adaptive Scheduling

Besides a single-chunk repair, PivotRepair also addresses a full-node repair that restores all lost chunks of a failed node. The full-node repair in PivotRepair is not a trivial task, as it triggers multiple single-chunk repairs which may incur competition for bandwidth resources, thereby impairing the repair performance or disturbing the foreground jobs.

A straightforward way is to perform Algorithm 1 for all single-chunk repairs and find the optimal parallelization of all schemes to fully utilize the bandwidth resources. However, a full-node repair often involves a large number of stripes, so it is impractical to check all stripes in advance. More importantly, even if we only check part of the related stripes and design optimal bandwidth-utilized repair pipelining methods, these methods may not work optimally after a while under rapidly changing available bandwidths in hot storage.

Alternatively, PivotRepair proposes an adaptive scheduling strategy for its full-node repair. The main idea is to arrange appropriate tasks to perform in different situations based on currently available bandwidths; in other words, we should avoid starting a new task when its pipelined repair tree's links are shared by too many repair tasks in progress, or it will cause competitions with these running tasks.

To this end, PivotRepair starts a new repair task based on a recommendation value (denoted by r); the larger r of a single-chunk repair task candidate (denoted by T_c) is, the more likely PivotRepair will start T_c. Clearly, r is mainly dominated by currently available bandwidths and running tasks, so we let

$$r=B_{min}-\sum_{i=1}^{n}\left[S_{(i,c)}\cdot(\alpha\cdot\frac{max(A_i-E_i,0)}{E_i}+\beta)\right], \quad (3)$$

where (i) B_{min}, as defined in §IV-B, is the minimum bandwidth of the pipelined repair tree of T_c under currently available bandwidths, (ii) n is the number of currently running tasks, (iii) $S_{(i,c)}$ shows the similarity degree of trees between T_c and the i^{th} running tasks T_i, where the similarity degree is calculated via the number of identical upload/download nodes between T_i and T_c, (iv) E_i is the expected time to finish the i^{th} running task based on its B_{min} obtained by Algorithm 1, (v) A_i is the actual time of performing the i^{th} running task, and (vi) α and β are parameters to indicate how strong the running tasks do not recommend T_c as a new task, since the larger α and β are, the smaller r is.

From Equation (3), we find that (i) when B_{min} is larger, r is also larger, which means we are more likely to recommend T_c that can be repaired with better pipelined repair performance; (ii) when n is larger, r becomes smaller, which means we are less likely to recommend any new task when there are more running tasks; (iii) when $S_{(i,c)}$ is larger, r becomes smaller, which means we are less likely to recommend T_c that has more identical links of the running tasks; (iv) when $\frac{max(A_i-E_i,0)}{E_i}$ is larger, r becomes smaller, which means we are less likely to recommend any new task when running tasks are more delayed. Here, $max(A_i - E_i, 0)$ is the delayed time of the i^{th} running task.

In this way, the value r can show the bandwidth competitions from both foreground jobs and repair jobs, and indicates whether a repair task should be performed at this time. As a result, we can check r of all tasks and choose the most recommended one to run every time to avoid the congestion as much as possible.

Specifically, the strategy works as follows: it first generates the pipelined trees of all stripes to be repaired via Algorithm 1 to compute r, and then it selects the stripe with the largest r (called best stripe) to perform the repair. Next, it repeats the operations until the value of r of the best stripe is smaller than the threshold that we fix based on experience from current tasks, which suggests that we should not add any repair task, due to too many running tasks or emerging foreground traffic. Thus PivotRepair obtains a couple of the best stripes that can be repaired in parallel currently. After one of the recently-added tasks finishes, it follows the above procedure to schedule again. Additionally, when the foreground jobs become continuously active, it will prevent new repair tasks from running and check periodically until available bandwidths turn sufficient, to avoid potential congestion.

Note that for each best stripe, PivotRepair always selects the node that has the most downlink bandwidth as the requestor, so all requestors are often distributed and the best stripes are repaired across multiple nodes. Thus, PivotRepair can be easily employed in large-scale systems to repair an entire node's stripes across different requestors as well as dozens of helpers according to their available bandwidths.

F. Discussion

Multi-chunk repair: PivotRepair mainly focuses on speeding up the repair of a single failed chunk per stripe, which accounts for the most repair scenarios in practice [24], [42] (e.g., over 98% of cases [42]). When a stripe has multiple failed chunks, PivotRepair resorts to conventional repair, where a node downloads k available chunks to regenerate all failed chunks.

Computation overhead: PivotRepair schedules repair operations mainly based on the available network capacity of nodes, but the computational resource usage may also need to be taken into account. One simple way to address the computation overhead issue is to check the computation capacity states of all nodes and identify which nodes have enough CPU cycles. We then run Algorithm 1 only based on the selected nodes. We may also partition time into timeslots, each of which only schedules a fraction of slice-repair tasks across nodes [51].

Multi-layer network: PivotRepair considers the case that all nodes are directly connected to a single switch. However, in modern data center networks, multi-layer network topologies are common and nodes may reside in different racks and be connected to different rack switches. Thus, the available bandwidth in cross-rack links is typically lower than that in the same rack. To address the topology heterogeneity, we can construct the PivotRepair's pipelining tree such that the pipelined repair can be performed locally within racks as much as possible. We pose this issue as future work.

V. Evaluation

A. Implementation

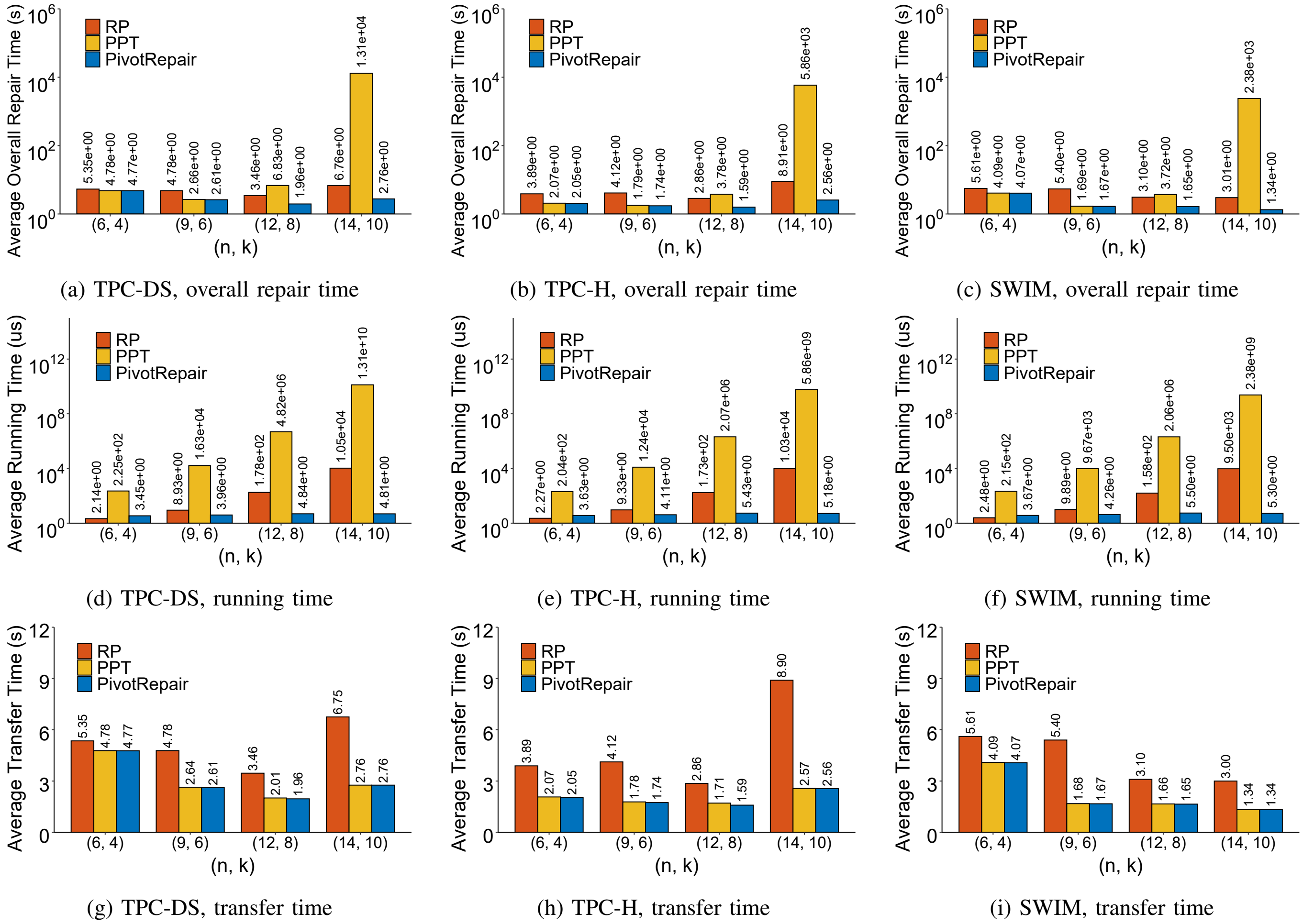
We implement a prototype system for PivotRepair in C++ and Python with about 3500 SLoCs, based on Intel ISA-L [2]. The system architecture has a single `Master` and multiple `Data-Nodes`, where the `Master` organizes k helpers to perform the repair while the `Data-Nodes` that store data serve as helpers. When receiving a repair request, the `Master` will call the algorithm to generate a repair scheme with the instant bandwidths situation, and send tasks to `Data-Nodes` to perform a pipelined repair. Note that we also implement the state-of-the-art repair techniques RP [28] and PPT [12] in the same system for fair comparisons.

B. Experiments Setup

We set Reed-Solomon codes with parameters (n,k) including: $(6,4)$ (a typical RAID-6 setting), $(9,6)$ (used in QFS [35]), $(12,8)$ (used in Baidu Atlas [26]), and $(14,10)$ (used by Facebook [43], [47]). Each chunk is set to 64 MiB [21] by default in coding.

We conduct cloud experiments under different network bandwidth environments based on the three workloads. Recall that we have 6000 records of the link bandwidth distribution of 16 different nodes for each of TPC-DS, TPD-H, and SWIM (§III-A). Here we randomly select a set of bandwidths situations with congestions for the three workloads respectively.

We evaluate performance of single-chunk and full-node repairs for PivotRepair, RP, and PPT in the three workloads. To replay network conditions of the workloads and distributions, we use the Linux traffic control command `tc` [8] to replay the link bandwidth changes of each node exactly the same as that in the workloads and distributions.



(a) TPC-DS, overall repair time (b) TPC-H, overall repair time (c) SWIM, overall repair time

(d) TPC-DS, running time (e) TPC-H, running time (f) SWIM, running time

(g) TPC-DS, transfer time (h) TPC-H, transfer time (i) SWIM, transfer time

Figure 5: Experiments 1-3: Overall repair time, running time, and transfer time for different traces and parameters (n,k).

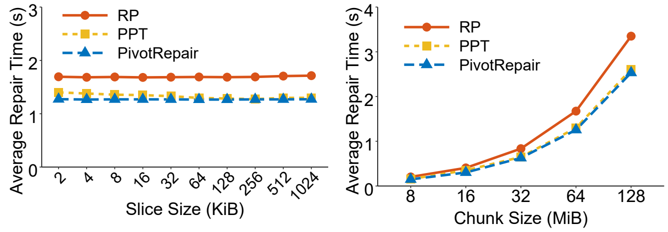
We conduct experiments on Amazon EC2 [1], deployed on 16 `m5.xlarge` instances in US East (North Virginia) region. We configure 15 instances that act as `Data-Nodes` (each represents a helper) and one instance that acts as `Master`. We report average results of each experiment over five runs.

C. Experiments

Experiment 1 (Overall single-chunk repair time): We evaluate the overall repair time, the algorithm running time and the transfer time for single-chunk repairs for different traces and parameters (n,k), where the overall repair time is composed of the algorithm running time and the transfer time. Figure 5 compares the average overall single-chunk repair time for PivotRepair, RP, and PPT. Compared to RP, PivotRepair is always faster in all cases, especially for the cases with larger k. For example, in Figure 5(b) with $k=10$, PivotRepair's overall repair time is reduced by 71.27% compared to RP. Compared to PPT, PivotRepair has a similar performance when n is small (e.g., $k=4$ and 6). For example, in Figure 5(c) with $k=6$, PivotRepair's and PPT's overall repair times are 1.67 s and 1.69 s, respectively. However, when k becomes larger, PPT's overall repair time grows exponentially. For instance, in Figure 5(a), PPT's overall single-chunk repair time increases from 6.83 s under $(12,8)$ to 1.31×10^4 s under $(14,10)$, which is much higher than RP and PivotRepair. The reason is that PPT takes a long time to generate its repair scheme when k is large, which will be discussed in Experiment 2.

Experiment 2 (Algorithm running time): We measure the algorithm running time of generating repair schemes for PivotRepair, RP, and PPT in three workloads. Figures 5(d)-5(f) show that among all traces, PivotRepair has larger running times to RP under $(6,4)$, but smaller ones under $(9,6)$, $(12,8)$ and $(14,10)$. Nevertheless, RP's running time in $(14,10)$ is about 10 ms, which is still affordable for a single-chunk repair. In contrast, PPT's running time increases exponentially with k, which ranges from 2.38×10^9 to 1.31×10^{10} s under $(14,10)$ for the three traces, which fails to handle the rapidly-changing bandwidths in hot storage. The reason is that PPT needs to enumerate all possible trees to search for the optimal one, which increases exponentially as k becomes larger. Finally, we see that PivotRepair's running time grows slowly and it only takes 4.81-5.30 μs to finish in $(14,10)$, which conforms to its $O(n\log n)$ time complexity (§IV-B).

Experiment 3 (Transfer time for single-chunk repair): We evaluate the transfer time for single-chunk repair in three



(a) Impact of slice size (b) Impact of chunk size

Figure 6: Experiments 4-5: Repair time for various sizes of slice and chunk.

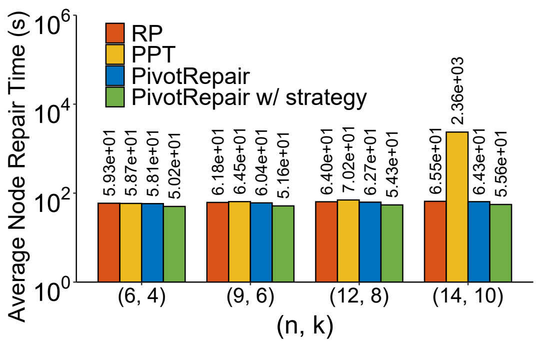


Figure 7: Experiment 6: Node repair time for different approaches.

workloads. Figures 5(g)-5(i) show that PivotRepair always remains as fast as PPT and keeps its performance gains over RP in three workloads. For example, in Figure 5(b) with $k=10$, PivotRepair reduces the single-chunk repair transfer time of RP by 71.2%. The low transfer time of PivotRepair conforms to the fact that PivotRepair's repair scheme has the optimal B_{min} (§IV-B).

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. Figure 6(a) shows that performances of all approaches keep steady with varied slice sizes, and we can draw a similar conclusion to the one in $(6,4)$ of Experiment 1, on the comparison between PivotRepair and the others.

Experiment 5 (Impact of chunk size): We evaluate the overall single-chunk repair time versus the chunk size, also with a fixed bandwidth situation. We set the slice size to 32 KiB and (n,k) is set to $(6,4)$. We vary the slice size from 8 MiB to 128 MiB. Figure 6(b) shows the overall single-chunk repair time of PivotRepair, RP, and PPT, which increase linearly with the chunk size, while PivotRepair also maintains its advantage.

Experiment 6 (Node Repair): We evaluate the node repair rate versus different (n,k). To perform a full-node repair, we first write a number of stripes of chunks randomly across all 15 nodes in the EC2 cluster, then erase 64 chunks of one node from 64 stripes to mimic a single node failure, and then repair all the erased chunks with different approaches. Figure 7 shows that PivotRepair outperforms the other schemes, which demonstrates its advantage in hot storage. Additionally, PivotRepair's adaptive scheduling strategy effectively reduces its original node repair time. For example, in Figure 7 under (9,6), PivotRepair's adaptive scheduling time (51.6 s) can be reduced up to 16.50% compared to RP (61.8 s). Note that the reduction between PivotRepair and RP of the full-node repair time is lower than that of the single-chunk repair time (see Experiment 1). The reason is that the full-node repair lasts for a longer period of time. Only a portion of which will have congestion (Figure 3), thereby degrading the advantages of PivotRepair. In addition, we find that PPT's full-node repair performance drops drastically when $k=10$, due to the same reason in Experiment 2.

VI. Related Work

Regenerating codes [19] are a family of erasure codes that minimize the repair traffic by allowing nodes to send encoded data for repair, including Product-Matrix codes [44], Zigzag codes [49], FMSR codes [23], PM-RBT codes [41], Butterfly codes [36], and Clay codes [50]. Some erasure codes minimize I/O during repair by sending fewer chunks in a single-node repair, such as Rotated RS codes [25] and Hitchhiker [43]. Locally repairable codes [24], [37], [47] mitigate repair I/O with extra storage. PivotRepair operates on RS codes, which satisfy linearity and are widely deployed in production (§II-A).

Previous studies propose repair-efficient techniques for erasure-coded storage. Lazy recovery [48] delays immediate repairs, so as to reduce the repair traffic at the expense of degraded reliability. PPR [33] reduces the single-chunk repair time by parallelizing the repair operation as partial operations. RP [28] further reduces the single-chunk repair time by pipelining the repair operation in slices (its extended version [30] also addresses hierarchical topologies and multi-chunk repair). PPT [12] improves RP by utilizing a pipelined tree. SMFRepair [55] uses idle nodes to bypass low-bandwidth links in the heterogeneous network. ECWide [22] exploits combined locality to address the wide-stripe repair problem. OpenEC [29] designs a directed-acyclic-graph-based programming abstraction to provide a unified and configurable framework for the erasure-coded storage system. HACFS [54] and Dayu [51] cope with dynamic workload changes by switching between different erasure codes [54] or scheduling repair tasks in free timeslots [51]. RepairBoost [32] focuses on improving full-node repair by careful traffic balancing and scheduling. PivotRepair aims to accelerate the repair operation in the hot storage, which can effectively construct an optimal pipelined repair tree via exploiting uncongested nodes.

VII. Conclusions

We propose PivotRepair, a fast pipelined repair technique for erasure-coded hot storage. PivotRepair is based on our observations that the repair job can be bottlenecked by rapidly-changing congested nodes in hot storage, while the storage network often contains uncongested nodes. We present an optimal algorithm to construct quickly the pipelined repair tree by exploiting uncongested nodes called pivots. We also propose an adaptive scheduling strategy to improve full-node repair performance. We prototype and evaluate PivotRepair on Amazon EC2. Our evaluation demonstrates the efficiency of PivotRepair in single-chunk and full-node repairs.

APPENDIX

Proof of Lemma 2: We prove via mathematical induction, i.e., checking the optimality when the i^{th} step of insertion is finished.

When $i = 1$, the only node in the priority queue Q is R (i.e., the requestor), and the only way to obtain the tree (denoted by T_1) is to insert N_1 as a child of R. Also R will become the only non-leaf node, so we get

$$min\{S_{nl}\} = prac(R), \text{where } prac(R) = down(R) \tag{4}$$

Thus, T_1 can reach optimal $min\{S_{nl}\}$.

For mathematical induction, we assume that tree $T_k(1 < k \leq n-1)$, constructed after k steps of insertion, can reach optimal $min\{S_{nl}\}$. Next, we examine the case when $i = k+1$, i.e., inserting N_i as a child of N_j, where N_j is the head of the priority queue. Here we define possible $prac_i$ if inserting a child to node N_i as $pp(i)$, such that we can obtain N_j that has the biggest pp in all nodes of T_k. Thus, we can deduce

$$min\{S_{nl}(T_{k+1})\} = min\{S_{nl}(T_k), pp(j)\} \tag{5}$$

We discuss the value of $pp(j)$ based on two cases.

Case 1: $pp(j) \geq min\{S_{nl}(T_k)\}$, i.e.,

$$min\{S_{nl}(T_{k+1})\} = min\{S_{nl}(T_k)\}. \tag{6}$$

Here T_{k+1} cannot achieve a higher $min\{S_{nl}\}$ by adjusting the tree, or it will contradict the assumption that T_k has optimal $min\{S_{nl}(T_k)\}$. So T_{k+1} achieves optimal $min\{S_{nl}(T_{k+1})$ in this case.

Case 2: $pp(j) < min\{S_{nl}(T_k)\}$, i.e.,

$$min\{S_{nl}(T_{k+1})\} = pp(j). \tag{7}$$

This case can be proved similar to Case 1 by contradiction and specified as follows. Note that the case 2 is based on discussion of the node to adjust, and we further discuss the adjustment on leaf nodes and non-leaf nodes based on two cases.

Case 2.1: For leaf nodes in T_{k+1}, we assume that swapping a non-leaf nodes in T_{k+1} with a leaf node N_t can generate a tree with a higher $min\{S_{nl}(T_{k+1})\}$. But we find that when N_t becomes a non-leaf node, it actually makes S_{nl} lower as N_t has a lower *prac* value. Thus this swap contradicts the assumption above, and there is no adjustment available on leaf nodes to improve $min\{S_{nl}(T_{k+1})\}$.

Case 2.2: For non-leaf nodes in T_k, we assume that moving children of a non-leaf node N_a to another node N_b can generate a tree with a higher $min\{S_{nl}(T_{k+1})\}$. But we find it impossible to improve the bottleneck, since the nodes excluding N_j do not have a higher pp value than $pp(j)$ due to the priority queue, and then increasing the number of N_b's children will make $pp(b)$ lower than $pp(j)$, which brings a lower bottleneck instead. Further, even if N_b is N_j, it is essentially equivalent to swapping the positions of the new child to be inserting N_i and one of N_a's children. This swap does not improve $pp(j)$, since it will not change the number of children for each node in T_{k+1}. Therefore, the moving contradicts the assumption above, and there is no adjustment available on non-leaf nodes to improve $min\{S_{nl}(T_{k+1})\}$.

Based on Cases 2.1 and 2.2, we conclude that $min\{S_{nl}(T_{k+1})\}$ is optimal in Case 2.

Based on Cases 1 and 2, T_{k+1} can achieve optimal $min\{S_{nl}(T_{k+1})\}$, i.e., the $k+1^{th}$ step establishes.

Based on above established steps, we can deduce that the n^{th} step still establishes, and thus tree T_n reaches optimal $min\{S_{nl}\}$. □

Proof of Lemma 3: Based on the replacing step, we can have that the l leaf nodes in T will be replaced by the top l nodes with the largest up among the unselected candidates and the previous leaf nodes in T, so as to get the final tree (denoted by T^*). Based on Lemma 1, we know that B_{min} is the smaller value of $min\{S_{nl}\}$ and $min\{S_l\}$. And based on Lemma 2, T has optimal $min\{S_{nl}\}$. While the above replacing operations do not change the number of each non-leaf node' children as well as S_{nl}, T^* still achieves optimal $min\{S_{nl}\}$.

We then discuss the situation of leaf nodes in T^* based on two cases.

Case 1 ($min\{S_l\} \geq min\{S_{nl}\}$): The up bandwidths in leaf nodes are not the bottleneck (to achieve $min\{S_l\}$), so no matter how the leaf nodes are distributed, $B_{min} = min\{S_{nl}\}$ is true, as we always keep the number of each non-leaf node' children unchanged based on the replacing step. If we assume that T^* can be adjusted to achieve a higher B_{min}, then it contradicts the deduction of Lemma 2 above. As a result, T^* reaches optimal $min\{S_l\}$ in this case.

Case 2 ($min\{S_l\} < min\{S_{nl}\}$): The leaf node with the lowest up bandwidth becomes the bottleneck, and $B_{min} = min\{S_l\}$ is always true no matter how the leaf nodes are distributed, as we always keep the number of each non-leaf nodes' children unchanged based on the replacing step.

Next, we discuss whether $min\{S_l\}$ can be improved by replacing the leaf node N_s that has the lowest up bandwidth in T^* based on two cases.

Case 2.1: If $min\{S_l\}$ of T^* can be improved via replacing with an unselected node, according to the replacing step, any leaf node in T^* has a higher up bandwidth than any node does not belong to T^*, making it impossible to improve $min\{S_l\}$ via replacing with any node that has a higher up bandwidth.

Case 2.2: If $min\{S_l\}$ of T^* can be improved via with another non-leaf node, then to keep the number of nodes in T^* unchanged, N_s has to become a new non-leaf node, which makes

$$min\{S_{nl}\} \leq theo(s) \leq up(s). \tag{8}$$

Based on Lemma 1 and Equation (8), we have

$$\begin{aligned} B_{min} &= min\{min\{S_{nl}\}, min\{S_l\}\} \\ &\leq theo(s) \leq up(s). \end{aligned} \tag{9}$$

We have that it cannot improve B_{min}.

Based on Cases 2.1 and 2.2, there is no available way to improve B_{min} in Case 2.

Based on Cases 1 and 2, T^* achieves optimal B_{min}. $\square$

References

[1] Amazon Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2/.

[2] Intel ISA-L. https://github.com/intel/isa-l.

[3] nload. https://linux.die.net/man/1/nload.

[4] On-line transactions and web content. https://www.backblaze.com/blog/whats-the-diff-hot-and-cold-data-storage.

[5] OpenStack Swift Object Storage Service. http://swift.openstack.org.

[6] Quick decision making. https://searchstorage.techtarget.com/definition/hot-data.

[7] SWIM. https://github.com/SWIMProjectUCB/SWIM.

[8] tc. https://linux.die.net/man/8/tc.

[9] TPC-DS. http://www.tpc.org/tpcds/.

[10] TPC-H. http://www.tpc.org/tpch/.

[11] M. K. Aguilera. Geo-distributed storage in data centers. Technical report, 2013.

[12] Y. Bai, Z. Xu, H. Wang, and D. Wang. Fast recovery techniques for erasure-coded clusters in non-uniform traffic network. In *Proc. of ICPP*, pages 61:1–61:10, 2019.

[13] Á. Björck and V. Pereyra. Solution of vandermonde systems of equations. *Mathematics of computation*, 24(112):893–903, 1970.

[14] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proc. of ACM SOSP*, pages 143–157. ACM, 2011.

[15] H. Chen, H. Zhang, M. Dong, Z. Wang, Y. Xia, H. Guan, and B. Zang. Efficient and available in-memory KV-store with hybrid erasure coding and replication. *ACM Trans. on Storage (TOS)*, 13(3):25, 2017.

[16] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. In *Proc. of ACM VLDB Endowment*, 2012.

[17] Y. L. 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*, pages 539–551, 2017.

[18] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. of ACM SIGCOMM*, 2013.

[19] 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)*, 56(9):4539–4551, Sep 2010.

[20] 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.

[21] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. 2003.

[22] Y. Hu, L. Cheng, Q. Yao, P. P. C. Lee, W. Wang, and W. Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, pages 233–248, 2021.

[23] Y. Hu, X. Li, M. Zhang, P. P. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Trans. on Storage (TOS)*, 13(4):33, 2017.

[24] 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.

[25] O. Khan, R. C. Burns, J. S. 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*, page 20, 2012.

[26] 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 *Proc. of IEEE MSST*, pages 1–14, 2015.

[27] W. E. Leland, M. S. Taqqu, W. Willinger, and D. V. Wilson. On the self-similar nature of Ethernet traffic (extended version). *IEEE/ACM Trans. on networking (TON)*, 2(1):1–15, 1994.

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

[29] X. Li, R. Li, P. P. C. Lee, and Y. Hu. Openec: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, pages 331–344, 2019.

[30] X. Li, Z. Yang, J. Li, R. Li, P. P. C. Lee, Q. Huang, and Y. Hu. Repair pipelining for erasure-coded storage: Algorithms and evaluation. *ACM Trans. on Storage (TOS)*, 17(2):29, 2021.

[31] 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*, pages 44–58. ACM, 2019.

[32] S. Lin, G. Gong, Z. Shen, P. P. Lee, and J. Shu. Boosting full-node repair in erasure-coded storage. In *Proc. of USENIX ATC*, 2021.

[33] 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*, page 30. ACM, 2016.

[34] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, et al. f4: Facebook's Warm BLOB Storage System. In *Proc. of USENIX OSDI*, pages 383–398, 2014.

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

[36] 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*, pages 81–94, 2016.

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

[38] 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*, pages 2801–2805. IEEE, 2012.

[39] J. S. Plank. Erasure codes for storage systems: A brief primer. *Login: The USENIX Magzine*, 38(6):44–50, 2013.

[40] H. Qiu, C. Wu, J. Li, M. Guo, T. Liu, X. He, Y. Dong, and Y. Zhao. Ec-fusion: An efficient hybrid erasure coding framework to improve both application and recovery performance in cloud storage systems. In *Proc. of IEEE IPDPS*, pages 191–201. IEEE, 2020.

[41] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *Proc. of USENIX FAST*, pages 81–94, 2015.

[42] 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.

[43] K. Rashmi, N. B. 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*, 2014.

[44] K. V. Rashmi, N. B. Shah, and P. V. Kumar. Optimal Exact-regenerating Codes for Distributed Storage at the MSR and MBR Points via a Product-matrix Construction. *IEEE Transactions on Information Theory (TIT)*, 57(8):5227–5239, Aug 2011.

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

[46] J. K. Resch and J. S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proc. of USENIX FAST*, 2011.

[47] 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*, pages 325–336, 2013.

[48] 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*, pages 1–7. ACM, 2014.

[49] I. Tamo, Z. Wang, and J. Bruck. Zigzag Codes: MDS Array Codes with Optimal Rebuilding. *IEEE Transactions on Information Theory*, 59(3):1597–1616, 2013.

[50] 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*, page 139, 2018.

[51] Z. Wang, G. Zhang, Y. Wang, Q. Yang, and J. Zhu. Dayu: fast and low-interference data recovery in very-large storage systems. In *Proc. of USENIX ATC*, pages 993–1008, 2019.

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

[53] 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*, pages 307–320. USENIX Association, 2006.

[54] M. Xia, M. Saxena, M. Blaum, and D. Pease. A tale of two erasure codes in HDFS. In *Proc. of USENIX FAST*, pages 213–226, 2015.

[55] H. Zhou, D. Feng, and Y. Hu. Multi-level forwarding and scheduling repair technique in heterogeneous network for erasure-coded clusters. In *Proc. of ICPP*, pages 19:1–19:11, 2021.