2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

Fast Predictive Repair in Erasure-Coded Storage

Zhirong Shen, Xiaolu Li, and Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong

***Abstract*—Erasure coding offers a storage-efficient redundancy mechanism for maintaining data availability guarantees in large-scale storage clusters, yet it also incurs high performance overhead in failure repair. Recent developments in accurate disk failure prediction allow soon-to-fail (STF) nodes to be repaired in advance, thereby opening new opportunities for accelerating failure repair in erasure-coded storage. To this end, we present a fast predictive repair solution called FastPR, which carefully couples two repair methods, namely migration (i.e., relocating the chunks of an STF node) and reconstruction (i.e., decoding the chunks of an STF node through erasure coding), so as to fully parallelize the repair operation across the storage cluster. FastPR solves a bipartite maximum matching problem and schedules both migration and reconstruction in a parallel fashion. We show that FastPR significantly reduces the repair time over the baseline repair approaches via mathematical analysis, large-scale simulation, and Amazon EC2 experiments.**

I. Introduction

Failures are prevalent in large-scale storage clusters and manifest at disks or various storage components [10], [15], [30], [36]. Field studies report that disk replacements in production are more frequent than estimated by vendors [30], [36], and latent sector errors are commonly found in modern disks [4].

To maintain data availability guarantees in the face of failures, practical storage clusters often stripe data with redundancy across multiple nodes via either replication or erasure coding. Replication creates identical data copies and is adopted by earlier generations of storage clusters, yet it incurs substantial storage overhead, especially with today's tremendous growth of data storage. On the other hand, erasure coding creates a limited amount of redundant data through coding computations, and provably maintains the same level of fault tolerance with much less storage redundancy than replication [41]. Today's large-scale storage clusters increasingly adopt erasure coding to provide low-cost fault-tolerant storage (e.g., [2], [10], [13], [26], [28]), and reportedly save petabytes of storage compared to replication [13], [26].

While being storage-efficient, erasure coding incurs high repair penalty. As an example, we consider Reed-Solomon (RS) codes [34], which are a popular erasure coding construction used in production [2], [10], [26], [28]. At a high level, RS codes encode k data chunks into n coded chunks for some parameters k and $n > k$, such that any k out of n coded chunks can reconstruct (or decode) all original k data chunks. However, repairing a lost chunk of RS codes needs to retrieve k available chunks for decoding, implying that both bandwidth and I/O costs for a single-chunk repair are amplified k times; in contrast, in replication, repairing a lost chunk can be simply done by retrieving another available chunk copy.

The high repair penalty is a fundamental issue in all erasure coding constructions: the repair traffic increases as the storage redundancy decreases [8]. Thus, there have been extensive studies on improving the repair performance of erasure coding, such as proposing theoretically proven erasure codes that minimize the repair traffic or I/Os during repair (e.g., [8], [13], [35]), or designing repair-efficient techniques that apply to all practical erasure codes including RS codes (e.g., [5], [20], [21], [24], [37], [38]). Conventional repair approaches are *reactive*, meaning that a repair operation is triggered only *after* a node failure is detected. Nevertheless, if we can predict impending failures in advance, we may *proactively* repair the lost data of any impending failed node to mitigate the repair penalty *before* any actual failure occurs.

Recent studies show that machine learning can achieve accurate prediction of disk failures in production environments with thousands of disks [6], [18], [23], [42], [43], [45]; in some cases, the prediction accuracy can even reach at least 95% [6], [18], [23], [45] with a very small false alarm rate. Motivated by the potential of highly accurate disk failure prediction, we can accurately pinpoint a soon-to-fail (STF) node and accelerate a repair operation by coupling two repair methods: (i) *migration*, in which we relocate the currently stored chunks of the STF node to other healthy nodes, and (ii) *reconstruction*, in which we reconstruct (or decode) the chunks of the STF node by retrieving the chunks of all healthy nodes in a storage cluster as in conventional reactive repair approaches. Migration addresses the bandwidth and I/O amplification issues that are inherent in erasure coding, while reconstruction exploits the aggregate bandwidth resources of all healthy nodes. An open question is how to carefully couple both migration and reconstruction so as to maximize the repair performance.

We present FastPR, a **Fast P**redictive **R**epair approach that carefully couples the migration and reconstruction of the chunks of the STF node, with the primary objective of minimizing the total repair time. FastPR schedules both migration and reconstruction of the chunks of the STF node in a parallel fashion, so as to exploit the available bandwidth resources of the underlying storage cluster. We address two repair scenarios: *scattered repair*, which stores the repaired chunks of the STF node across all other existing nodes in the storage cluster, and *hot-standby repair*, which stores the repaired chunks of the STF node in dedicated hot-standby nodes. We present an in-depth study of FastPR through mathematical analysis, large-scale simulation, and Amazon EC2 experiments, and make the following contributions:

- We first present mathematical analysis on the optimal predictive repair in minimizing the total repair time. We show

978-1-7281-0057-9/19/$31.00 ©2019 IEEE
DOI 10.1109/DSN.2019.00062

that it can reduce the repair time by over 30% compared to the conventional reactive repair.

- We present FastPR, which is designed to parallelize both migration and reconstruction across the storage cluster. We formulate a bipartite maximum matching problem and design polynomial repair algorithms to schedule both migration and reconstruction to be effectively executed in a parallel fashion.
- We implement a FastPR prototype in C++ and show that it can be seamlessly integrated with HDFS of Hadoop 3.1.1 [3] without changing the HDFS codebase.
- We conduct large-scale simulation based on our modified FastPR prototype. FastPR significantly reduces the repair times of both migration-only and reconstruction-only approaches (the latter is equivalent to the conventional reactive repair), for example, by 62.7% and 40.6%, respectively for $(n, k) = (16, 12)$ in scattered repair. Also, FastPR is close to the optimal point (different by no more than 11.4%) that is derived from our mathematical analysis.
- We deploy FastPR on HDFS and conduct testbed experiments on Amazon EC2 with 25 instances, so as to demonstrate the effectiveness of FastPR in real-world environments. FastPR still significantly reduces the repair times of both migration-only and reconstruction-only approaches, for example, by up to 42.6% and 71.7%, respectively for different parameters of (n, k) in scattered repair.

The source code of FastPR is available for download at: **http://adslab.cse.cuhk.edu.hk/software/fastpr**.

II. Background and Problem

A. Erasure Coding

We consider a storage cluster that stores file data over a network of *nodes* (which may refer to disks or servers). Files are stored as a collection of fixed-size *chunks* that are striped across different nodes. The chunk size is typically on the order of MBs (e.g., 64 MB or larger) to mitigate the disk I/O overhead. We encode the chunks with erasure coding to achieve low-redundancy fault tolerance.

In this paper, we focus on RS codes [34], which are a well-known erasure code construction and have been widely deployed in production [2], [10], [26], [28]. We can construct an RS code, denoted by RS(n, k), with two parameters n and k, where $k < n$. RS(n, k) encodes k uncoded chunks into n coded chunks of the same size via linear combinations based on Galois Field arithmetic, such that any k out of n coded chunks can reconstruct (or decode) the original k uncoded chunks; in other words, it can tolerate the loss of any $n - k$ coded chunks. Each collection of n coded chunks is called a *stripe* and is distributed across n distinct nodes, so as to tolerate any $n-k$ node failures. In practice, a storage cluster stores multiple stripes that are independently encoded and distributed across different sets of n distinct nodes. Note that we can construct RS codes in *systematic* form, meaning that k of the n coded chunks of a stripe are exactly the k original uncoded chunks that can be directly accessed in normal mode. Nevertheless, in this paper, we do not differentiate whether the chunks of a stripe are in uncoded or coded form; instead, we collectively refer to them as "chunks" in the following discussion.

RS codes are popular mainly for two reasons. First, RS codes are *storage-optimal* (a.k.a. *Maximum Distance Separable (MDS)* in coding theory), meaning that RS(n, k) achieves the minimum amount of redundancy (i.e., n/k times the original data) while allowing any k chunks of a stripe to reconstruct the original data. Second, RS codes are *general*, as they can support any general parameters n and k (provided that $k < n$). However, RS codes incur substantial *repair traffic* (i.e., the amount of available data being retrieved for repairing the lost data). Specifically, during a single node failure, repairing any lost chunk under RS(n, k) needs to first retrieve k chunks of the same stripe from k surviving nodes for decoding, implying that the amount of repair traffic is k times that of lost data.

Many repair-efficient erasure codes have been proposed to reduce the repair bandwidth over RS codes, while preserving the same or slightly higher redundancy. For example, Minimum-Storage Regenerating (MSR) codes [8] are storage-optimal as RS codes, and minimize the repair traffic for repairing a single lost chunk by allowing surviving nodes to send the linear combinations of the locally stored data during repair. Recently proposed MSR codes (e.g., PM-RBT [32], Butterfly [29], and Clay [40]) further eliminate the need of computing linear combinations by reading directly the required sub-chunks from surviving nodes during repair. Locally repairable codes (LRCs) [13], [35] trade slightly higher redundancy for improved repair performance by storing an extra local coded chunk associated with a subset of chunks (i.e., a local group) of a stripe, so that repairing a single lost chunk can be done by retrieving the available chunks within the same local group. Note that the amount of repair traffic, even though being minimized, remains larger than the amount of lost data [8], so the bandwidth and I/O amplification issues still exist during repair.

While we focus on RS codes, our methodologies also apply to repair-efficient codes, which retrieve available data from k' healthy nodes (e.g., $k < k' \leq n - 1$ in MSR codes, or $k' < k$ in LRCs) when repairing a lost chunk, such that the amount of repair traffic is less than the total size of k chunks. We provide an example for LRCs in Section III.

B. Predictive Repair

Existing erasure codes (including RS codes and other repair-efficient codes) take a *reactive* repair approach and trigger repair operations upon detecting a lost chunk (or a node failure). In this work, we explore a proactive approach, namely *predictive repair*, to predict a soon-to-fail (STF) node and restore its currently stored chunks to other healthy (i.e., non-STF) nodes in advance before it actually fails or is replaced.

Motivation: Modern disk vendors adopt SMART (Self-Monitoring, Analysis and Reporting Technology) to collect statistics on different disk reliability aspects. Each disk includes a SMART tool in its microprocessor firmware to monitor disk operations and report a number of *SMART attributes* (e.g., error counts, disk temperature, power-on hours, etc.). If the SMART attributes of interest are above some thresholds, the

disk firmware triggers failure warnings [14]. For example, RAIDShield [22] replaces a potentially failed disk whose reallocated sector count from SMART is above a threshold, and such protection is reportedly deployed in production. Note that SMART attributes are arguably inaccurate indicators of failed disks [11], [30] (e.g., over half of the failed disks do not show SMART errors [30]). Nevertheless, machine learning has recently been shown to effectively predict disk failures in production environments based on SMART data [6], [18], [23], [42], [45] and additional system events [43]. In addition to disk failures, machine learning is proven effective to predict the failures of other types of components (e.g., machines or switches) in data center environments [17], [44].

Repair methods: We design the predictive repair mechanism by coupling two methods, namely *migration* and *reconstruction*, to repair the chunks of an STF node. We discuss the pros and cons of both methods.

Migration reads the stored chunks directly from an STF node and relocates them to one or multiple healthy nodes. It does not introduce extra traffic compared to normal reads, and hence has no bandwidth and I/O amplification issues. However, the performance is bottlenecked by the available bandwidth of the STF node.

On the other hand, reconstruction follows the conventional reactive repair, by retrieving multiple chunks (e.g., k chunks in RS(n, k)) from healthy nodes to reconstruct the chunks of the STF node. Since multiple stripes are typically spread across the storage cluster, we can exploit the available bandwidth resources of the storage cluster and involve all healthy nodes to participate in the repair of multiple chunks of the STF node in a parallel fashion. However, the drawback is that it introduces extra traffic.

Goal and assumptions: Our idea is to take advantage of both migration and reconstruction to maximize the predictive repair performance, with the primary objective of *minimizing the repair time of repairing a single STF node*. Minimizing the repair time is critical for reducing the window of vulnerability, especially when failures are correlated and subsequent failures appear sooner after the first failure [36].

Our work makes the following assumptions. First, we assume that there is at most one STF node at a time in the storage cluster, based on the observation that single-node repair is the most dominant repair event (e.g., 98% of the total [33]) as opposed to multi-node repair [13], [33]. Nevertheless, if multiple failed nodes occur within a stripe, we can resort to the conventional reactive repair. Second, to mitigate the risk of data loss, we assume that proactively repairing the chunks of the STF node is necessary, even though the STF node is a false alarm and is later deemed healthy after extensive operational tests [36]. Third, we assume that the chunks stored in the STF node remain accessible during repair until the STF node actually fails or is shut down for replacement. Finally, the chunk distribution may become imbalanced after multiple repairs, and we assume that the storage cluster periodically rebalances the chunk distribution in the background.

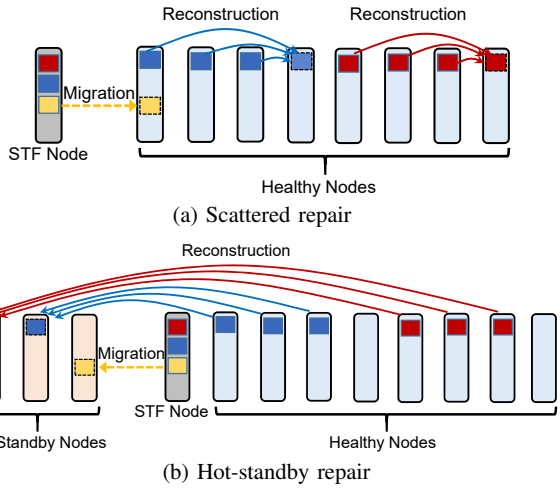


Fig. 1. Repair scenarios: (a) scattered repair and (b) hot-standby repair. Chunks of the same color belong to the same stripe. Different stripes are spread across the storage cluster.

C. Repair Scenarios

We study how we apply predictive repair in two scenarios, namely *scattered repair* and *hot-standby repair*, which specify different destination nodes for storing repaired chunks of an STF node. Figure 1 illustrates both scenarios.

Scattered repair selects existing healthy nodes in the storage cluster to repair and store the repaired chunks of the STF node. Specifically, to repair a chunk of each stripe, scattered repair should choose the healthy node that currently does not store any chunk of the same stripe, so that we maintain the same degree of (node-level) fault tolerance. Thus, the performance of scattered repair depends on the cluster scale and the distribution of the stripes across the cluster.

On the other hand, hot-standby repair deploys one or multiple dedicated nodes (called *hot-standby nodes*) to repair and store the repaired chunks of the STF node. Before repair, such hot-standby nodes serve as backup nodes without participating in normal applications, but take over the service of the STF node after repair. The performance of hot-standby repair depends on the number of hot-standby nodes available for repair.

III. Mathematical Analysis

We conduct simple mathematical analysis to provide preliminary insights into the performance gain of the optimal predictive repair over the conventional reactive repair in a large-scale storage cluster.

General formulation: We first provide a general formulation that addresses the performance of both reactive and predictive repair strategies in both scattered and hot-standby repair scenarios, and later extend the formulation for different scenarios. Let M be the total number of nodes in a storage cluster and U be the total amount of chunks of the STF node that are repaired. Let x be the amount of chunks being repaired by migration; hence, $U-x$ is the amount of chunks being repaired by reconstruction.

For migration, let t_m be the time to migrate a chunk from the STF node to another healthy node. Thus, the total time spent in migration is $x \cdot t_m$.

For reconstruction, let t_r be the time to repair a chunk of the STF node. Recall that we reconstruct each chunk of the STF node by retrieving k chunks from k healthy nodes under RS(n, k). If $M-1$ (i.e., the number of healthy nodes in the storage cluster) is significantly larger than k, then we can reconstruct multiple chunks of the STF node simultaneously. Suppose that we divide the reconstruction process into multiple rounds, such that in each round, we can find $G \leq \frac{M-1}{k}$ non-overlapping groups of k nodes that belong to different stripes and retrieve the chunks from them in parallel (i.e., k chunks from each group). Thus, we can repair G chunks of the STF node in time t_r through reconstruction, and the total time spent in reconstruction is $\frac{U-x}{G} \cdot t_r$.

Let $T(x)$ be the total repair time of predictive repair. As both migration and reconstruction are performed in parallel, we have:

$$T(x) = \max(x \cdot t_m, \frac{U-x}{G} \cdot t_r). \quad (1)$$

We can readily show that $T(x)$ is minimized when $x \cdot t_m = \frac{U-x}{G} \cdot t_r$, or equivalently, $x = \frac{U \cdot t_r}{G \cdot t_m + t_r}$. Thus, the minimum predictive repair time (denoted by T_P) is:

$$T_P = \frac{U \cdot t_r \cdot t_m}{G \cdot t_m + t_r}. \quad (2)$$

On the other hand, as the reactive repair simply follows reconstruction without migration, the total repair time of reactive repair (denoted by T_R) is $T(0)$ from Equation (1), i.e.,

$$T_R = \frac{U \cdot t_r}{G}. \quad (3)$$

Modeling of repair scenarios: We now extend our general formulation to address both scattered and hot-standby repair scenarios. Our goal is to model the values of t_m and t_r.

In our modeling, we decompose a repair operation into three steps carried out in a sequential manner: (i) *read* (i.e., reading chunks from the underlying local storage), (ii) *transmission* (i.e., transmitting the chunks over the network), and (iii) *write* (i.e., writing the repaired chunks into new existing nodes (for scattered repair) or hot-standby nodes (for hot-standby repair)). Let b_d and b_n be the disk and network bandwidths, respectively, and c be the chunk size. We calculate the read time, transmission time, and write time for each repaired chunk.

To simplify our analysis, we do not address disk I/O interference, which occurs in the following cases: (i) in scattered repair, an existing healthy node reads a locally stored chunk while writing the repaired chunk for another stripe, and (ii) in hot-standby repair, a hot-standby node writes the chunks from both migration and reconstruction. We also assume that the computational costs of coding operations are negligible compared to disk I/Os and network transmission [16].

We first model t_m in migration, which applies to both scattered and hot-standby repairs. The read time, transmission time, and write time for each repaired chunk are c/b_d, c/b_n, and c/b_d, respectively. Thus,

$$t_m = \frac{c}{b_d} + \frac{c}{b_n} + \frac{c}{b_d}. \quad (4)$$

We now model t_r in reconstruction. For the scattered repair, each of the k healthy nodes can read the chunks of a stripe in parallel, so the read time for each repaired chunk is c/b_d. Each repaired chunk triggers k chunks transmitted over the network, so the transmission time for each repaired chunk is $k \cdot c/b_n$. The write time for each repair chunk is c/b_d. Thus,

$$t_r = \frac{c}{b_d} + \frac{k \cdot c}{b_n} + \frac{c}{b_d} \quad \text{(for scattered repair)}. \quad (5)$$

For the hot-standby repair, let h be the number of hot-standby nodes, such that $h \ll G$; hence, the transmissions and writes to the hot-standby nodes are the bottlenecks. Recall that in each round, we can repair G chunks of the STF node in parallel, and they trigger a total of $G \cdot k$ chunks transmitted over the network. Thus, each hot-standby node on average receives $\frac{G \cdot k}{h}$ chunks from the network and writes $\frac{G}{h}$ repaired chunks. Thus,

$$t_r = \frac{c}{b_d} + \frac{G \cdot k \cdot c}{h \cdot b_n} + \frac{G \cdot c}{h \cdot b_d} \quad \text{(for hot-standby repair)}. \quad (6)$$

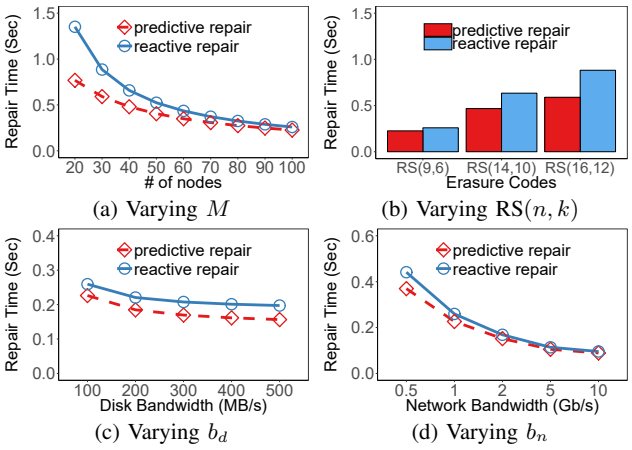
Analysis: We study via mathematical analysis the performance gain of the optimal predictive repair (Equation (2)) over the conventional reactive repair (Equation (3)). In our analysis, we assume that we can find the maximum of $G = \frac{M-1}{k}$ non-overlapping groups of chunks to repair $\frac{M-1}{k}$ chunks of the STF node at time t_r in parallel.

We consider the following default configurations. We set $M = 100$, $U =$ 1,000 chunks of size $c = 64$ MB each, $b_d = 100$ MB/s, and $b_n = 1$ Gb/s. We consider RS$(9, 6)$, the default erasure coding configuration in QFS [28]. For hot-standby repair, we set $h = 3$. We vary one of the parameters and analyze its performance impact. Here, we measure the repair time per chunk.

Figure 2 first shows the repair time in scattered repair. Predictive repair shows a higher performance gain than reactive repair when the number of nodes is small (Figure 2(a)), k is large (Figure 2(b)), b_d is large (Figure 2(c)), and b_n is small (Figure 2(d)). The reason is that the repair penalty due to the amplified repair traffic in reactive repair becomes more significant in such cases. Overall, predictive repair reduces the repair time of reactive repair in all cases, for example, by 33.1% in RS$(16, 12)$ (Figure 2(b)).

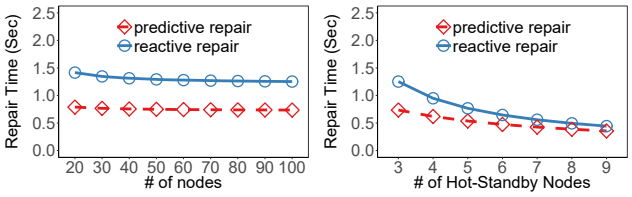
Figure 3 shows the repair time in hot-standby repair. When the number of hot-standby nodes h is small, predictive repair is more significant (Figure 3(b)). For example, when $h = 3$, predictive repair reduces the repair time by 41.3%.

Extension for LRCs: We elaborate how we can generalize the above analysis for LRCs [13], [35], which divide k chunks into l local groups (assuming that k is divisible by l) and associate each local group with a local coded chunk (Section II-A). Repairing a single lost chunk is done by retrieving $k' = \frac{k}{l}$ chunks within the local group. Thus, we can retrieve the chunks



(a) Varying M (b) Varying RS(n, k) (c) Varying b_d (d) Varying b_n

Fig. 2. Mathematical analysis in scattered repair.



(a) Varying M (b) Varying h

Fig. 3. Mathematical analysis in hot-standby repair.

from $G' \leq \frac{M-1}{k'}$ non-overlapping groups in each round of reconstruction. By substituting G with G' in Equation (2) and k with k' in Equations (5) and (6), our analysis follows.

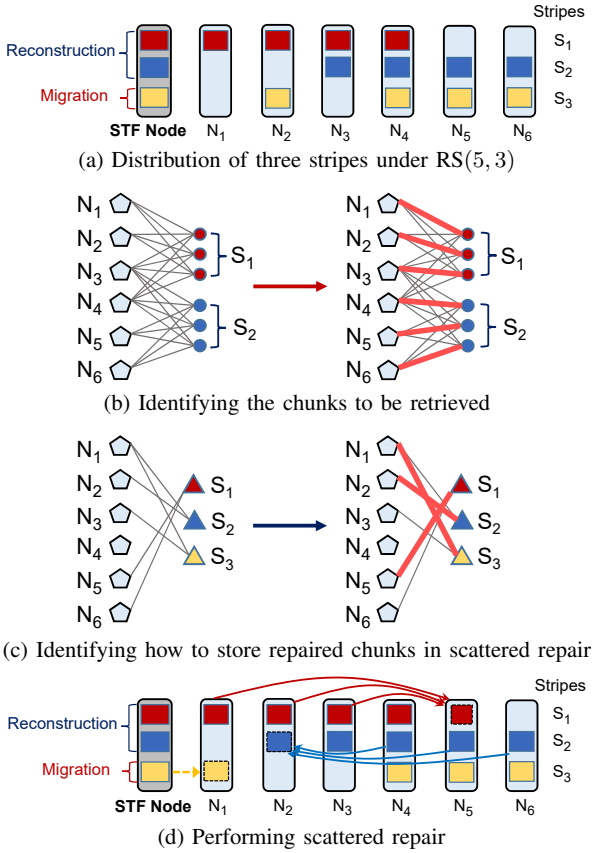
IV. Fast Predictive Repair

We now present the design of FastPR, which couples both migration and reconstruction to achieve fast predictive repair. FastPR aims to identify, with polynomial complexity, a repair solution that minimizes the repair time (Section III).

The main idea of FastPR is to decompose a repair operation of an STF node into multiple *repair rounds* that are iteratively executed. Each repair round comprises the sets of chunks of an STF node that are to be repaired through either migration or reconstruction. FastPR examines the chunk distribution and identifies the appropriate sets of chunks for migration or reconstruction in each repair round, such that it minimizes the total number of repair rounds and hence the repair time.

A. Design Overview

We first provide an overview of how FastPR performs both migration and reconstruction in a repair round. Figure 4 depicts the idea under RS$(5, 3)$ (i.e., $n = 5$ and $k = 3$). Let N_i be the i-th healthy node, and S_i be the stripe for the i-th chunk of the STF node to be repaired. Suppose that we are given the sets of c_m and c_r chunks of an STF node for migration and reconstruction in a repair round, respectively. For example, the storage cluster (with $M = 7$ nodes) in Figure 4(a) has $M - 1 = 6$ healthy nodes $N_1, \cdots, N_6$, while the chunks of the STF node correspond to stripes S_1, S_2, and S_3. Also, we have $c_m = 1$ and $c_r = 2$. We address the following two issues.



(a) Distribution of three stripes under RS$(5, 3)$

(b) Identifying the chunks to be retrieved

(c) Identifying how to store repaired chunks in scattered repair

(d) Performing scattered repair

Fig. 4. Overview of how FastPR performs both migration and reconstruction in a repair round. Chunks of the same color belong to the same stripe.

First, given the c_r chunks of the STF node to be repaired through reconstruction, FastPR needs to identify the $k \cdot c_r$ chunks to be retrieved from $k \cdot c_r$ healthy nodes. We formulate the selection of the $k \cdot c_r$ chunks as a bipartite maximum matching problem (Figure 4(b)). Specifically, we construct a bipartite graph with the left and right sets of vertices, in which the left set contains $M-1$ *node vertices* representing all $M-1$ healthy nodes, and the right set contains $k \cdot c_r$ *chunk vertices* representing the $k \cdot c_r$ chunks to be retrieved from c_r stripes (i.e., k chunks per stripe). We add an edge from a node vertex to a chunk vertex if the corresponding node stores a chunk for the corresponding stripe. For example, in Figure 4(b), the node vertices for $\{N_1, N_2, N_3, N_4\}$ and $\{N_3, N_4, N_5, N_6\}$ are connected to the $k = 3$ chunk vertices for stripes S_1 and S_2, respectively (i.e., each chunk vertex is connected to $n-1$ node vertices). Our goal is to find a maximum matching with $k \cdot c_r$ edges, implying that all $k \cdot c_r$ chunk vertices are included in the maximum matching. For example, the maximum matching in Figure 4(b) states that we can retrieve chunks from N_1, N_2, and N_3 to reconstruct the chunk for S_1. Such a maximum matching can be solved, say, as a maximum flow problem by Ford-Fulkerson algorithm in $O(VE)$ time, where V and E are the numbers of vertices and edges, respectively [7].

Second, FastPR needs to identify how to store the $c_m + c_r$

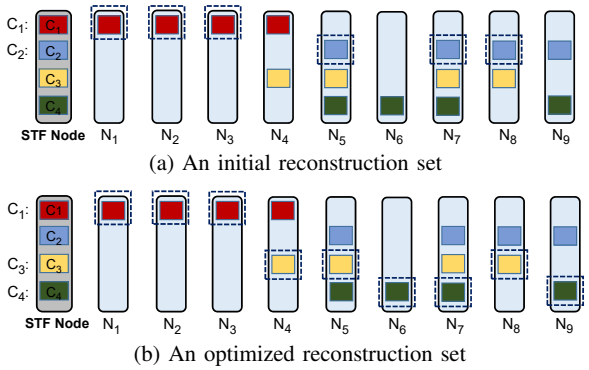
repaired chunks. For scattered repair, we store the repaired chunks in $c_m + c_r$ existing healthy nodes, such that the node-level fault tolerance is maintained (i.e., any $n-k$ node failures are tolerable). We again formulate the selection of the $c_m + c_r$ existing nodes as a bipartite maximum matching problem. Specifically, we construct a bipartite graph with the left and right sets of vertices, in which the left set contains $M-1$ *node vertices* representing all $M-1$ healthy nodes, and the right set contains $c_m + c_r$ *stripe vertices* representing the $c_m + c_r$ stripes being repaired. We add an edge from a node vertex to a stripe vertex if the node *does not* store a chunk for the stripe before the repair operation. For example, in Figure 4(c), the node vertices for $\{N_5, N_6\}$, $\{N_1, N_2\}$, and $\{N_1, N_3\}$ are connected to the stripe vertices for stripes S_1, S_2, and S_3, respectively. Thus, each stripe vertex is connected to $M-1-(n-1) = M-n$ node vertices (recall that $M-1$ is the number of healthy nodes in the storage cluster, and $n-1$ is the number of healthy nodes that store the chunks of the corresponding stripe). If M is sufficiently large such that $M-n \geq c_m + c_r$, then any subset of $c_m + c_r$ stripe vertices are connected to at least $M-n \geq c_m + c_r$ node vertices. By Hall's Theorem [7], we can always find a maximum matching that includes all $c_m + c_r$ stripe vertices. Such a maximum matching determines the node where each repaired chunk is stored. For example, the maximum matching in Figure 4(c) shows that we can store the repaired chunk of S_1 in N_5. For hot-standby repair, we simply evenly distribute the repaired chunks to all h hot-standby nodes.

Given the sets of chunks for migration and reconstruction in a repair round, FastPR performs both migration and reconstruction in parallel (see Figure 4(d) for scattered repair). In the following, we show how we identify the chunks for migration and reconstruction in each repair round.

B. Finding Reconstruction Sets

Design idea: To minimize the total number of repair rounds, FastPR aims to maximize the number of chunks of the STF node to be repaired in each repair round. Suppose that all chunks in the STF node are to be repaired through the reconstruction method. We partition all chunks of the STF node into *reconstruction sets*. Each chunk in a reconstruction set can be repaired through reconstruction by retrieving k chunks from k healthy nodes, such that at most one chunk is retrieved from each of the $M-1$ healthy nodes. Intuitively, a reconstruction set contains the chunks of the STF node that can be reconstructed in parallel in a single repair round. To improve parallelism, a reconstruction set should contain as many chunks of the STF node as possible (at most $\frac{M-1}{k}$ as shown in Section III), or equivalently, FastPR should return as few reconstruction sets as possible that cover all the chunks of the STF node to be repaired.

To find reconstruction sets, we build on the bipartite maximum matching problem in Section IV-A to check if a subset of chunks of the STF node can be reconstructed in parallel. Based on the reconstruction sets, we then schedule the chunks to be repaired through either migration or reconstruction (Section IV-C).



(a) An initial reconstruction set

(b) An optimized reconstruction set

Fig. 5. Finding reconstruction sets: (a) an initial reconstruction set that can only reconstruct two chunks in parallel; (b) an optimized reconstruction set that can reconstruct three chunks in parallel. Chunks with dashed boxes are those retrieved from healthy nodes for reconstruction.

Algorithm details: Algorithm 1 presents how finding reconstruction sets works. Let $\mathcal{C}$ denote the set of all chunks of the STF node to be repaired, and $\mathcal{R}$ denote a reconstruction set.

Algorithm 1 finds a reconstruction set $\mathcal{R}$ through the FIND function (Lines 9-40). First, we form an initial reconstruction set $\mathcal{R}$ (Lines 10-17), by incrementally adding a chunk in $\mathcal{C}$ to $\mathcal{R}$ and checking if the new set of chunks in $\mathcal{R}$ can be reconstructed in parallel. Specifically, for each chunk $C_i \in \mathcal{C}$, we call the MATCH function (Lines 1-8) to form a bipartite graph that contains $M-1$ node vertices representing the healthy nodes and $k(1+|\mathcal{R}|)$ chunk vertices representing the chunks for $\mathcal{R} \cup \{C_i\}$ (Line 2). Then we find the maximum matching on the bipartite graph (Line 3). If the maximum matching has $k(1+|\mathcal{R}|)$ edges, it implies that the chunks in $\mathcal{R} \cup \{C_i\}$ can be reconstructed through the chunks at $k(1+|\mathcal{R}|)$ different healthy nodes in parallel. In this case, the MATCH function returns true (Lines 4-6). We add C_i to $\mathcal{R}$ and also remove C_i from $\mathcal{C}$ (Lines 14-15).

Given the initial reconstruction set $\mathcal{R}$, we check if we can expand $\mathcal{R}$ by swapping one of its chunks with another chunk that is currently not in $\mathcal{R}$ (Lines 18-38). Specifically, for each $C_i \in \mathcal{R}$ and $C_j \in \mathcal{C}$ (note that $\mathcal{C}$ now contains the residual chunks that are to be repaired), we swap them to form a new reconstruction set $\mathcal{R}'$. We check if adding any chunk $C_l \in \mathcal{C}$ to $\mathcal{R}'$ can expand the maximum matching; if so, we include C_l into some dummy set $\mathcal{A}_{i,j}$ (Lines 20-31). Finally, we find the pair (i^*, j^*) such that $|\mathcal{A}_{i^*,j^*}|$ is maximum (Line 32), so as to add the most chunks into $\mathcal{R}$. If $|\mathcal{A}_{i^*,j^*}| > 0$, we swap $C_{i^*} \in \mathcal{R}$ and $C_{j^*} \in \mathcal{C}$, add $\mathcal{A}_{i^*,j^*}$ to $\mathcal{R}$, and remove $\mathcal{A}_{i^*,j^*}$ from $\mathcal{C}$ (Lines 33-35); otherwise, we cannot further expand $\mathcal{R}$, so we break the while-loop (Line 36).

Finally, in the main procedure (Lines 41-48), we repeatedly call the FIND function on $\mathcal{C}$, until all chunks in $\mathcal{C}$ are organized into reconstruction sets. We return the collection of all reconstruction sets $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_d\}$, where d is the total number of reconstruction sets.

Example: Figure 5 shows an example of finding all reconstruc-

Algorithm 1 Finding Reconstruction Sets

```
1: function MATCH(R, C_i)
2:     Form a bipartite graph based on M − 1 nodes and R ∪ {C_i}
3:     Find a maximum matching on the bipartite graph
4:     if the maximum matching has k(1 + |R|) edges then
5:         return true
6:     end if
7:     return false
8: end function
9: function FIND(C)
10:    // Form an initial reconstruction set
11:    Initialize R = ∅
12:    for each chunk C_i ∈ C do
13:        if MATCH(R, C_i) equals true then
14:            Set R = R ∪ {C_i}
15:            Set C = C − {C_i}
16:        end if
17:    end for
18:    // Optimize the reconstruction set
19:    while true do
20:        for each chunk C_i ∈ R do
21:            for each chunk C_j ∈ C do
22:                Initialize A_{i,j} = ∅
23:                Set R' = R ∪ {C_j} − {C_i}
24:                for each chunk C_l ∈ C do
25:                    if MATCH(R', C_l) equals true then
26:                        Set A_{i,j} = A_{i,j} ∪ {C_l}
27:                        Set R' = R' ∪ {C_l}
28:                    end if
29:                end for
30:            end for
31:        end for
32:        Set (i*, j*) = arg max_{(i,j)}{|A_{i,j}|}
33:        if |A_{i*,j*}| > 0 then
34:            Set R = R ∪ A_{i*,j*} ∪ {C_{j*}} − {C_{i*}}
35:            Set C = C ∪ {C_{i*}} − {C_{j*}} − A_{i*,j*}
36:        else break
37:        end if
38:    end while
39:    return (R, C)
40: end function
41: procedure MAIN(C)
42:    Initialize d = 0
43:    while C ≠ φ do
44:        Set d = d + 1
45:        (R_d, C) = FIND(C)
46:    end while
47:    return {R_1, ··· , R_d}
48: end procedure
```

tion sets, in which there are four stripes encoded by RS$(5, 3)$ that are stored in 10 nodes. Suppose that the STF node stores four chunks $\mathcal{C} = \{C_1, C_2, C_3, C_4\}$. Based on Algorithm 1, we obtain an initial reconstruction set $\mathcal{R} = \{C_1, C_2\}$, and we can verify that adding C_3 or C_4 to $\mathcal{R}$ cannot expand the maximum matching (Figure 5(a)). We further optimize $\mathcal{R}$ by checking if it can include more chunks in $\mathcal{C}$. By replacing C_2 in $\mathcal{R}$ with C_3 in $\mathcal{C}$, we see that C_4 can now be added to the reconstruction set, so the new reconstruction set becomes $\mathcal{R} = \{C_1, C_3, C_4\}$ (Figure 5(b)). The remaining C_2 forms another reconstruction set. Thus, we have two reconstruction sets $\{\mathcal{R}_1, \mathcal{R}_2\} = \{\{C_1, C_3, C_4\}, \{C_2\}\}$.

C. Repair Scheduling

Design idea: Given the reconstruction sets, we schedule how the chunks of the STF node are actually repaired through migration or reconstruction in a repair round. Our observation is that the chunks in a larger reconstruction set are more preferred to be repaired through reconstruction, since more chunks can be repaired in parallel. In contrast, the chunks in a smaller reconstruction set are more preferred to be repaired through migration to reduce the repair traffic.

We need to first decide how many chunks are migrated or reconstructed (i.e., c_m and c_r, respectively) in each repair round. Here, we set c_r as the number of chunks in a reconstruction set being selected to be reconstructed, and estimate c_m based on the disk bandwidth b_d and network bandwidth b_n. Specifically, t_m and t_r denote the times to repair a chunk through migration and reconstruction in a repair round, respectively (Section III). For t_m, we can derive it via Equation (4). For t_r, we derive it via Equations (5) and (6) for scattered and hot-standby repairs, respectively; note that $G = c_r$ here. Since t_r is also the reconstruction time in a repair round, we can calculate $c_m = \frac{t_r}{t_m}$, meaning that migrating c_m chunks spends the same amount of time as reconstructing c_r chunks in a repair round.

Algorithm details: Algorithm 2 presents how repair scheduling works, given the input of reconstruction sets. First, we sort all d reconstruction sets $\{\mathcal{R}_1, \cdots, \mathcal{R}_d\}$ by their numbers of chunks in monotonically descending order (Line 1). We initialize two indices $l = 1$ and $u = d$ to refer to the currently considered reconstruction sets that have the most and the fewest chunks (Line 2). First, we compute c_m from $c_r = |\mathcal{R}_l|$ (Line 4). If $|\mathcal{R}_{l+1} \cup \cdots \cup \mathcal{R}_u| \leq c_m$, it implies that $\mathcal{R}_{l+1} \cup \cdots \cup \mathcal{R}_u$ (denoted by $\mathcal{M}_l$) can be repaired through migration, in parallel with the reconstruction of $\mathcal{R}_l$. We break the while-loop and the algorithm completes (Lines 5-8).

Otherwise, we find the reconstruction sets with the fewest chunks such that they can be repaired through migration in a repair round. We find the largest x, where $\sum_{i=x}^{u} |\mathcal{R}_i| > c_m$ (Line 9). To fine-tune our selection, we further select a subset $\mathcal{R}'_x \subset \mathcal{R}_x$ (the chunks of $\mathcal{R}'_x$ are randomly selected from $\mathcal{R}_x$), where $|\mathcal{R}'_x| = c_m - \sum_{i=x+1}^{u} |\mathcal{R}_i|$, such that $\mathcal{R}'_x$ can also be repaired through migration (Lines 10-11). Finally, we set $\mathcal{M}_l = \mathcal{R}'_x \cup \mathcal{R}_{x+1} \cup \cdots \cup \mathcal{R}_u$, which are the c_m chunks to be repaired through migration (Line 12), in parallel with the reconstruction of $\mathcal{R}_l$ in the same repair round. We update l and u (Lines 13-14) and iterate for another repair round.

Example: Figure 6 gives an example on how we schedule the repair rounds. Suppose that we have $d = 7$ reconstruction sets. For simplicity, we fix $c_m = 4$ (note that for hot-standby repair, c_m may change across different repair rounds as it is a function of $G = c_r$). For the first repair round, we find that the largest x is five, such that $|\mathcal{R}_5| + |\mathcal{R}_6| + |\mathcal{R}_7| > c_m = 4$. We further find a subset $\mathcal{R}'_5 \subset \mathcal{R}_5$ with one chunk, such that $\mathcal{M}_1 = \mathcal{R}'_5 \cup \mathcal{R}_6 \cup \mathcal{R}_7$ is repaired through migration in parallel with the reconstruction of $\mathcal{R}_1$ in the first repair round. Note that the number of remaining chunks in $\mathcal{R}_5$ reduces to two. Finally, we can repair all chunks in three repair rounds.

Algorithm 2 Repair Scheduling

Input: Reconstruction sets
Output: Chunks of the STF node to be migrated and reconstructed in each repair round

1: Sort $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_d\}$, where $|\mathcal{R}_1| \geq |\mathcal{R}_2| \geq \cdots \geq |\mathcal{R}_d|$
2: Initialize $l = 1$ and $u = d$
3: **while true do**
4: Compute c_m from c_r, where $c_r = |\mathcal{R}_l|$
5: **if** $|\mathcal{R}_{l+1} \cup \cdots \cup \mathcal{R}_u| \leq c_m$ **then**
6: $\mathcal{M}_l = \mathcal{R}_{l+1} \cup \cdots \cup \mathcal{R}_u$
7: **break**
8: **end if**
9: Find the largest x, where $\sum_{i=x}^{u} |\mathcal{R}_i| > c_m$
10: Find a subset $\mathcal{R}'_x \subset \mathcal{R}_x$, where $|\mathcal{R}'_x| + \sum_{i=x+1}^{u} |\mathcal{R}_i| = c_m$
11: Set $\mathcal{R}_x = \mathcal{R}_x - \mathcal{R}'_x$
12: Set $\mathcal{M}_l = \mathcal{R}'_x \cup \mathcal{R}_{x+1} \cup \cdots \cup \mathcal{R}_u$
13: Set $l = l + 1$
14: Set $u = x$
15: **end while**

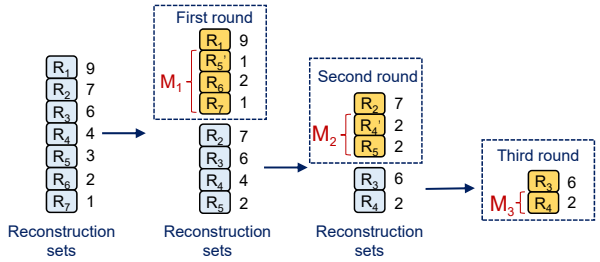


Fig. 6. Example of repair scheduling. Each number denotes the number of chunks in the corresponding reconstruction set.

D. Time Complexity Analysis of Algorithms 1 and 2

We first consider the MATCH function in Algorithm 1. MATCH aims to find the maximum matching on a candidate reconstruction set, whose size is at most $\frac{M-1}{k}$ (Section III). Thus, the resulting bipartite graph in MATCH has at most $\frac{M-1}{k} \cdot k = M - 1$ chunk vertices (together with $M - 1$ node vertices). Each chunk vertex connects to $n - 1$ node vertices, so there are at most $(M - 1)(n - 1)$ edges in the bipartite graph. Thus, MATCH finds a maximum matching in a bipartite graph in $O(M^2 n)$ time (Section IV-A).

We next consider the FIND function in Algorithm 1. Forming an initial reconstruction set (Lines 10-17) calls MATCH $|\mathcal{C}|$ times. Optimizing the reconstruction set (Lines 18-38) expands $\mathcal{R}$ no more than $\frac{M-1}{k}$ times (Line 34), and each time calls MATCH $|\mathcal{C}|^2|\mathcal{R}| \leq |\mathcal{C}^2 \frac{M-1}{k}|$ times (Lines 20-31). Thus, the time complexity of FIND is $O(|\mathcal{C}|^2 M^4 n)$.

Overall, Algorithm 1 calls FIND at most $|\mathcal{C}|$ time, so its time complexity $O(|\mathcal{C}|^3 M^4 n)$.

Finally, we analyze the time complexity of Algorithm 2. Sorting d reconstruction sets (Line 1) takes $O(d \log d)$ time. For each repair round, we scan d reconstruction sets to find the largest x (Line 9) in $O(d)$ time, and find the subset $\mathcal{R}'_x$ in $\mathcal{R}_x$ (Line 10) in $O(|\mathcal{R}_x|)$ time. Since the number of repair rounds is at most d, the complexity of Lines 3-15 is $O(d(d + |\mathcal{R}_x|))$. Also, as $d \leq |\mathcal{C}|$ and $|\mathcal{R}_x| \leq \frac{M-1}{k}$, the time complexity of Algorithm 2 is $O(|\mathcal{C}|(|\mathcal{C}| + \frac{M}{k}))$.

Discussion: Note that Algorithm 1, even with polynomial complexity, incurs high running time for large $|\mathcal{C}|$ and M. We suggest two options to mitigate the overhead. The first option is to partition the chunks of the STF node into *chunk groups* and find the reconstruction sets for each chunk group (which now becomes $\mathcal{C}$). Another option is that we can run Algorithm 1 for each possible STF node in advance and store the results when they are required [16]. We present microbenchmarks on Algorithm 1 in Section VI-B.

V. IMPLEMENTATION

We have built a FastPR prototype in C++, including the coding operations for chunk reconstruction using Jerasure v1.2 [31]. Our prototype has around 2,400 lines of code.

System architecture: FastPR comprises a *coordinator* and multiple *agents*, such that each agent is deployed in a storage node. The coordinator is responsible for instructing multiple agents to perform repair operations. It manages the metadata information of each chunk, including the location of the chunk and the identity of the stripe to which the chunk belongs. When the coordinator detects an STF node, it determines which chunk in the STF node will be repaired through migration or reconstruction across different repair rounds. For each repair round, the coordinator issues commands to the associated agents to start the repair operation. Upon receiving the commands from the coordinator, the agent in the STF node migrates chunks to other destination nodes, while the agents in the healthy nodes retrieve chunks from local storage and send them to the agents of the destination nodes for chunk reconstruction. The agents return acknowledgments to the coordinator upon completion, and the coordinator issues commands for the next repair round.

Integration with HDFS: We argue that FastPR can be seamlessly integrated with state-of-the-art distributed storage systems. As a case study, we run FastPR atop HDFS of Hadoop 3.1.1 [3]. Figure 7 illustrates the integration of FastPR with HDFS. Specifically, HDFS comprises a NameNode for storage management and multiple DataNodes for storing chunks. We deploy the coordinator on the NameNode and an agent in each DataNode. The coordinator on the NameNode accesses HDFS metadata by executing the command "`hdfs fsck / -files -blocks -locations`", through which the coordinator can determine the chunk location and the stripe information. The coordinator can then instruct the agents to start the repair operation. Note that each HDFS chunk is associated with a small metadata block (typically of size around 1 MB for a 128 MB HDFS chunk). In our deployment, FastPR migrates each metadata block from the STF node to a new DataNode.

After the repair operation, the DataNodes report the new locations of the repaired chunks to the NameNode through periodic heartbeats, and the NameNode updates the chunk and stripe information. We emphasize that our FastPR deployment requires *no* modification to the HDFS codebase.

Multi-threading: We further accelerate a repair operation via multi-threading. Specifically, we partition a chunk into multiple small equal-size units called *packets*. When a node sends a

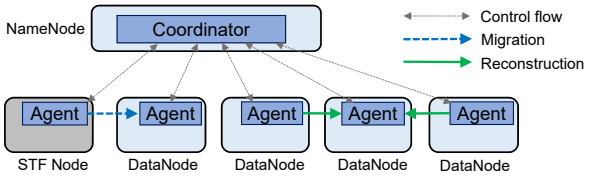


Fig. 7. Integration of FastPR with HDFS.

chunk to another node, it creates two threads that operate in units of packets in a pipelined manner: one thread for reading packets from the local storage, and another thread for sending packets over the network. In addition, when a node is about to store a repaired chunk, it creates one thread for decoding the received packets, as well as multiple threads for receiving packets from multiple nodes.

VI. Performance Evaluation

We evaluate FastPR in two aspects: (i) large-scale simulation and (ii) testbed experiments on Amazon EC2.

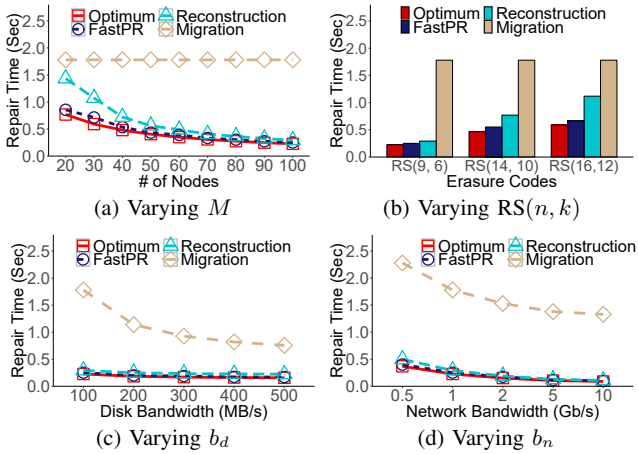
A. Simulation Experiments

We conduct simulation on FastPR to evaluate its performance in a large-scale storage cluster. We design a single-machine simulator for FastPR by modifying our prototype. In the simulator, we remove all the actual operations of disk I/Os and network transmission from the prototype, and simulate the operations by computing their execution times based on the input network and disk bandwidths. Note that the main algorithms, including finding reconstruction sets and repair scheduling, are still preserved.

We compare FastPR with three approaches: (i) *migration-only*, in which we directly migrate all the chunks of the STF node to other healthy nodes; (ii) *reconstruction-only*, in which we find the reconstruction sets based on Algorithm 1, but we repair each of them in a repair round by reconstruction only without calling Algorithm 2 (note that it corresponds to the conventional reactive repair); and (iii) *optimum*, which we derive from Equation (2) based on our modeled t_m and t_r (Section III).

We assume the following default configurations. We configure a storage cluster of $M = 100$ nodes with the disk bandwidth $b_d = 100$ MB/s and network bandwidth $b_n = 1$ Gb/s. We encode the chunks by RS$(9, 6)$ adopted by QFS [28], while we also consider RS$(14, 10)$ (adopted by Facebook [26]) and RS$(16, 12)$ (coding parameters used by Azure [13]). We fix the chunk size as 64 MB, and randomly distribute 1,000 stripes of chunks across the storage cluster. For hot-standby repair, we fix the number of hot-standby nodes $h = 3$. We vary one configuration parameter in our simulation experiments and evaluate its impact. We measure the repair time per chunk, averaged over 30 runs.

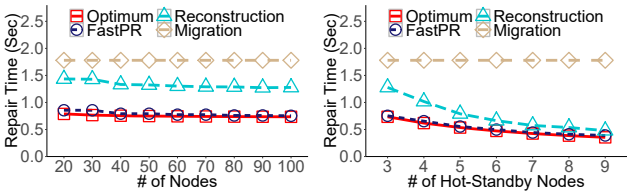
Experiment A.1 (Scattered repair): Figure 8 shows the simulation results of the repair time per chunk in scattered repair, in which we vary M, RS(n, k), b_d, and b_n. Migration-only is the worst among all approaches, since its performance



(a) Varying M (b) Varying RS(n, k)

(c) Varying b_d (d) Varying b_n

Fig. 8. Experiment A.1: Simulation results of repair time per chunk in scattered repair.



(a) Varying M (b) Varying h

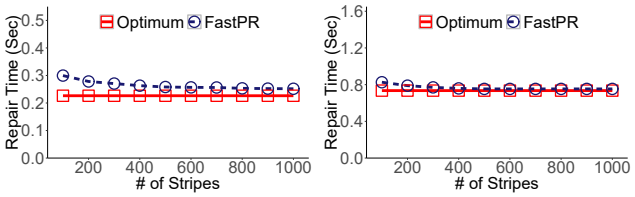
Fig. 9. Experiment A.2: Simulation results of repair time per chunk in hot-standby repair.

is bottlenecked by the STF node. Reconstruction-only has similar performance to FastPR since it can exploit the available bandwidth resources of the storage cluster as FastPR. However, it incurs higher repair time per chunk than FastPR when M is small (Figure 8(a)) or (n, k) is large (Figure 8(b)), since the available bandwidth resources of the storage cluster are more limited for smaller M and the repair traffic increases for larger k. Overall, FastPR reduces the repair times of both migration-only and reconstruction-only, for example, by 62.7% and 40.6% for RS$(16, 12)$ (Figure 8(b)).

In practice, FastPR deviates from the optimum since the number of chunks of the STF node that can be repaired in parallel depends on the chunk distribution. Nevertheless, our simulation results show that the repair time of FastPR is only 11.4% more than the optimum on average.

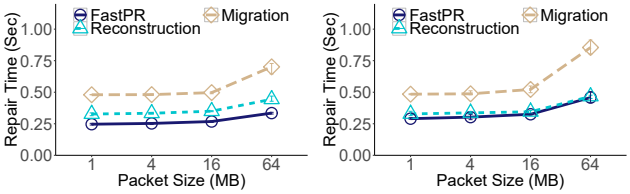
Experiment A.2 (Hot-standby repair): Figure 9 shows the simulation results of the repair time per chunk in hot-standby repair, in which we vary M and h. The repair performance is mainly bottlenecked by the hot-standby nodes, and the repair time has limited variance across different values of M (Figure 9(a)). When $h = 3$, FastPR reduces the repair times of migration-only and reconstruction-only by 57.7% and 41.0%, respectively. FastPR maintains high performance in hot-standby repair, and its repair time is only 5.4% more than the optimum on average.

Experiment A.3 (Impact of the number of stripes): Fig-



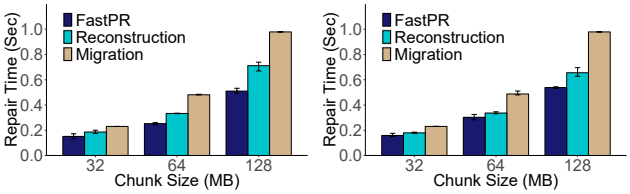
(a) Scattered repair (b) Hot-standby repair

Fig. 10. Experiment A.3: Simulation results of repair time per chunk versus the number of stripes.



(a) Scattered repair (b) Hot-standby repair

Fig. 11. Experiment B.1: Impact of the packet size.



(a) Scattered repair (b) Hot-standby repair

Fig. 12. Experiment B.2: Impact of the chunk size.

ure 10 shows the repair time per chunk versus the number of stripes distributed across the storage cluster. Here, we only focus on FastPR and the optimum to compare their differences. Increasing the number of stripes provides more flexibility of FastPR to identify the reconstruction sets that maximize parallelism. We observe that when the number of stripes is at least 400, the differences between FastPR and the optimum are very small (within 15%). This implies that we can limit our selection of reconstruction sets (i.e., Algorithm 1) to a smaller group of chunks to mitigate the running time overhead (Section IV-D), while achieving the near-optimal repair performance.

B. Testbed Experiments

We also conduct testbed experiments on Amazon EC2 to understand the performance of FastPR in a real-world cloud environment. We set up 25 virtual machine instances of type `m5.large` in the US East (North Virgina) region. Each instance runs Ubuntu 14.04.5 LTS, and is equipped with two vCPUs with 2.5 GHz Intel Xeon Platinum, 8 GB RAM, and 50 GB of EBS storage. Before running our experiments, we conduct preliminary measurements and find that each instance achieves 142 MB/s of disk bandwidth (on sequential writes) and 5 Gb/s of network bandwidth (measured by `iperf`). We deploy FastPR atop HDFS (Section V), in which we run the FastPR coordinator and the HDFS NameNode in one instance, and both a FastPR agent and an HDFS DataNode in each of 21 instances that serve as storage nodes. We reserve the remaining three instances for hot-standby repair (i.e., $h = 3$ hot-standby nodes).

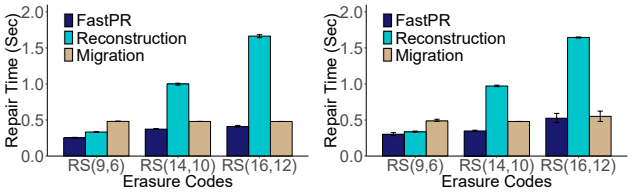
We assume the following default configurations. We configure the erasure coding scheme as RS$(9, 6)$, the chunk size as 64 MB, and the packet size as 4 MB. All instances use all available network bandwidth (5 Gb/s) for chunk transmission. We randomly distribute stripes across the storage cluster, such that the number of chunks in the STF node being repaired is fixed as 50 chunks in each experimental run for consistent benchmarking. We compare FastPR with migration-only and reconstruction-only (Section VI-A). We plot the average repair time per chunk over five runs, and also include the error bars showing the maximum and the minimum across the five runs (some may be invisible from the plots).

Experiment B.1 (Impact of the packet size): We first study the impact of multi-threading by evaluating the repair time per chunk versus the packet size, varied from 1 MB to 64 MB; note that for the packet size 64 MB, we do not enable multi-threading as it is equal to the default chunk size. Figure 11 shows that the repair time reduces for smaller packet sizes, as multi-threading can parallelize different steps of a repair operation. For example, when the packet size reduces from 64 MB to 4 MB, the repair time of FastPR reduces by 31.4% (the reduction is negligible when the packet size further reduces to 1 MB). For all packet sizes, FastPR reduces the repair times of migration-only and reconstruction-only by 37.7-52.3% and 1.9-24.7%, respectively.

Experiment B.2 (Impact of the chunk size): We now evaluate the impact of the chunk size, varied from 32 MB to 128 MB (where the packet size is fixed as 4 MB). Figure 12 shows that the repair time per chunk increases with the chunk size, yet FastPR still reduces the repair times of migration-only and reconstruction-only by 31.1-47.9% and 10.0-28.3% across all chunk sizes, respectively.

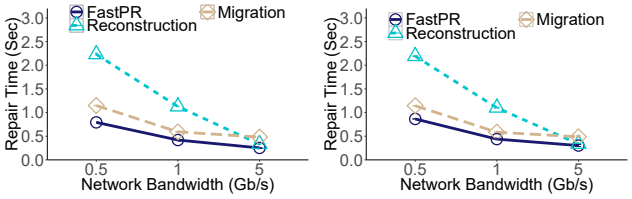
Experiment B.3 (Impact of different erasure codes): We now evaluate the repair time per chunk for different erasure codes. Here, we focus on RS$(9, 6)$, RS$(14, 10)$, and RS$(16, 12)$. Figure 13 shows the results. The performance of migration-only remains unaffected by different (n, k), yet the repair time of reconstruction-only increases significantly in RS$(14, 10)$ and RS$(16, 12)$ compared to RS$(9, 6)$, as it increases the amount of repair traffic. FastPR also sees higher repair time in RS$(14, 10)$ and RS$(16, 12)$, yet it still achieves the least repair time among all approaches. Overall, FastPR reduces the repair times of migration-only and reconstruction-only by 42.6% and 17.1% in RS$(9, 6)$, 24.9% and 63.4% in RS$(14, 10)$, and 9.6% and 71.7% in RS$(16, 12)$, respectively.

Experiment B.4 (Impact of network bandwidth): We study how the network bandwidth (i.e., b_n) affects the repair time. We use the Wonder Shaper tool [1] to control the network adapter bandwidth. Here, we vary b_n as 0.5 Gb/s, 1 Gb/s, and 5 Gb/s (the default one without bandwidth limiting). Figure 14 shows that



(a) Scattered repair (b) Hot-standby repair

Fig. 13. Experiment B.3: Impact of different erasure codes.



(a) Scattered repair (b) Hot-standby repair

Fig. 14. Experiment B.4: Impact of network bandwidth.

the repair time of reconstruction-only significantly increases when the network bandwidth is limited, as it incurs a high amount of repair traffic. Overall, FastPR reduces the repair times of migration-only and reconstruction-only by 27.7% and 62.5% when $b_n = 0.5$ Gb/s, and 27.1% and 61.5% when $b_n = 1$ Gb/s, respectively.

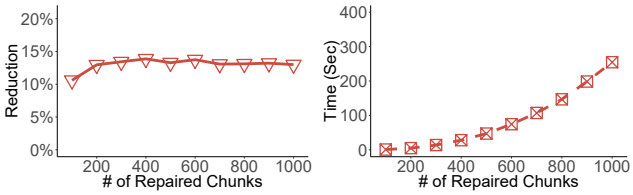
Experiment B.5 (Microbenchmarks): A key component of FastPR is to find the reconstruction sets (Algorithm 1 in Section IV-B). We study Algorithm 1 in two aspects.

First, we analyze the effectiveness of optimizing the selection of the initial reconstruction set in Algorithm 1 (i.e., Lines 18-38). We compare the numbers of reconstruction sets returned by Algorithm 1 with and without Lines 18-38, denoted by d_{opt} and d_{ini}, respectively. Intuitively, if $d_{opt} < d_{ini}$, the optimization step reduces the number of reconstruction sets returned and hence includes more chunks in each reconstruction set on average to exploit a higher degree of parallelism. Figure 15(a) shows the reduction of d_{opt} compared to d_{ini} versus the number of repaired chunks (i.e., $|\mathcal{C}|$), averaged over 30 runs. Overall, d_{opt} is 13% less than d_{ini}, while the reduction becomes fairly stable when the number of repaired chunks is at least 200.

Second, we measure the running time of Algorithm 1 in one of our Amazon EC2 instances. Figure 15(b) shows the running time of Algorithm 1 versus the number of repaired chunks, averaged over 30 runs. The running time increases from 0.84 s for 100 repaired chunks to 254.63 s for 1,000 repaired chunks. Nevertheless, we can run Algorithm 1 on the repaired chunks of the STF node by groups and pre-compute Algorithm 1 for each STF node offline to mitigate its overhead (Section IV-D).

VII. Related Work

Proactive fault tolerance: Our work builds on the potential of accurate failure prediction [6], [18], [23], [42], [43], [45], and takes one step further to improve repair performance in erasure-coded storage. Some studies take proactive approaches to enhance the fault tolerance of storage systems. Proactive



(a) Reduction of d_{opt} over d_{ini} (b) Running time of Algorithm 1

Fig. 15. Experiment B.5: Microbenchmarks.

replication [9], [39] injects redundant data before failures occur to avoid traffic bursts during repair when failures happen. RAIDShield [22] uses the reallocated sector count to identify STF disks and replaces them in advance. ProCode [19] leverages disk failure prediction results to store erasure-coded data in STF nodes with replication, so as to balance between recovery performance and storage efficiency. While FastPR also builds on accurate disk failure prediction, it does not introduce redundant data, but instead focuses on accelerating the repair of an STF node.

Reactive repair: Most repair approaches for erasure-coded storage are reactive and triggered only when failures actually happen. Some studies propose new erasure codes with less repair traffic (e.g., MSR codes [8], [29], [32], [40] and LRCs [13], [35]). We argue that our work applies to such new erasure codes (Section III). On the other hand, some studies design repair-efficient techniques that work for existing erasure codes, such as lazy repair [5], [38] or parallelizing partial repair operations [20], [21], [24], [37]. FastPR also targets existing erasure codes (RS codes in particular), but takes a predictive repair approach to improve repair performance.

To exploit all available bandwidth resources in repair, parity declustering [12], [25] distributes stripes across multiple nodes to parallelize repair operations. RAMCloud [27] applies a similar design for fast repair under replication-based storage (i.e., replicas are scattered across the entire system). As a comparison, FastPR focuses on maximizing the parallelism of repair operations under the parity-declustered layout.

VIII. Conclusion

This paper explores the potential of leveraging the accurate failure prediction of a storage cluster to minimize the total repair time of erasure-coded storage. We present FastPR, a fast predictive repair approach that repairs in advance the chunks of an STF node before it actually fails. Its core idea is to carefully couple both migration and reconstruction to fully parallelize a repair operation across the whole storage cluster. Our mathematical analysis, large-scale simulation, and Amazon EC2 experiments demonstrate that FastPR outperforms the conventional reactive repair approach that triggers repair operations only after a failed node is detected.

Acknowledgements: We thank our shepherd, Etienne Rivière, and the anonymous reviewers for their valuable comments. This work was supported by HKRGC (GRF 14216316 and CRF C7036-15G) and NSFC (61602120).

REFERENCES

[1] The Wonder Shaper 1.4. https://github.com/magnific0/wondershaper.

[2] Erasure Coding in Ceph. https://ceph.com/planet/erasure-coding-in-ceph/, 2014.

[3] Apache Hadoop 3.1.1. https://hadoop.apache.org/docs/r3.1.1/, 2018.

[4] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proc. of ACM SIGMETRICS*, 2007.

[5] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. M. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of USENIX NSDI*, 2004.

[6] M. M. Botezatu, I. Giurgiu, J. Bogojeska, and D. Wiesmann. Predicting Disk Replacement towards Reliable Data Centers. In *Proc. of ACM SIGKDD*, 2016.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2009.

[8] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.

[9] A. Duminuco, E. Biersack, and T. En-Najjary. Proactive Replication in Distributed Storage Systems Using Machine Availability Estimation. In *Proc. of ACM CoNEXT*, 2007.

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

[11] M. Goldszmidt. Finding Soon-to-fail Disks in a Haystack. In *Proc. of USENIX HotStorage*, 2012.

[12] M. Holland, G. A. Gibson, and D. P. Siewiorek. Architectures and Algorithms for On-line Failure Recovery in Redundant Disk Arrays. *Distributed Parallel Databases*, 2(3):295–335, 1994.

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

[14] G. F. Hughes, J. F. Murray, K. Kreutz-Delgado, and C. Elkan. Improved Disk-Drive Failure Warnings. *IEEE Trans. on Reliability*, 51(3):350–357, Sep 2002.

[15] W. Jiang, C. Hu, Y. Zhou, and A. Kanevsky. Are Disks the Dominant Contributor for Storage Failures?: A Comprehensive Study of Storage Subsystem Failure Characteristics. *ACM Trans. on Storage*, 4(3):7, 2008.

[16] 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*, 2012.

[17] Y.-L. Lee, D.-C. Juan, X.-A. Tseng, Y.-T. Chen, and S.-C. Chang. DC-Prophet: Predicting Catastrophic Machine Failures in DataCenters. In *Proc. of ECML-PKDD*, 2017.

[18] J. Li, X. Ji, Y. Jia, B. Zhu, G. Wang, Z. Li, and X. Liu. Hard Drive Failure Prediction Using Classification and Regression Trees. In *Proc. of IEEE/IFIP DSN*, 2014.

[19] P. Li, J. Li, R. J. Stones, G. Wang, Z. Li, and X. Liu. Procode: A Proactive Erasure Coding Scheme for Cloud Storage Systems. In *Proc. of IEEE SRDS*, 2016.

[20] R. Li, X. Li, P. P. C. Lee, and Q. Huang. Repair Pipelining for Erasure-Coded Storage. In *Proc. of USENIX ATC*, 2017.

[21] 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*, 2019.

[22] A. Ma, F. Douglis, G. Lu, D. Sawyer, S. Chandra, and W. Hsu. RAIDShield: Characterizing, Monitoring, and Proactively Protecting Against Disk Failures. In *Proc. of USENIX FAST*, 2015.

[23] F. Mahdisoltani, I. Stefanovici, and B. Schroeder. Proactive Error Prediction to Improve Storage System Reliability. In *Proc. of USENIX ATC*, 2017.

[24] 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*, 2016.

[25] R. R. Muntz and J. C. S. Lui. Performance Analysis of Disk Arrays under Failure. In *Proc. of VLDB*, Aug 1990.

[26] S. Muralidhar, W. Lloyd, S. Roy, et al. F4: Facebook's Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.

[27] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast Crash Recovery in RAMCloud. In *Proc. of ACM SOSP*, 2011.

[28] M. Ovsiannikov, S. Rus, D. Reeves, P. Sutter, S. Rao, and J. Kelly. The Quantcast File System. *Proc. of the VLDB Endowment*, 6(11):1092–1101, 2013.

[29] 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.

[30] E. Pinheiro, W.-D. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *Proc. of USENIX FAST*, 2007.

[31] J. Plank, S. Simmerman, and C. Schuman. Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications-Version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[32] 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*, 2015.

[33] K. V. 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 FacebookWarehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.

[34] I. S. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the society for industrial and applied mathematics*, 8(2):300–304, 1960.

[35] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. *Proc. of the VLDB Endowment*, 6(5):325–336, 2013.

[36] B. Schroeder and G. A. Gibson. Disk Failures in The Real World: What Does An MTTF of 1,000,000 Hours Mean to You? In *Proc. of USENIX FAST*, 2007.

[37] Z. Shen, J. Shu, and P. P. C. Lee. Reconsidering Single Failure Recovery in Clustered File Systems. In *Proc. of IEEE/IFIP DSN*, 2016.

[38] 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*, 2014.

[39] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. T. Morris, M. F. Kaashoek, and J. Kubiatowicz. Proactive Replication for Data Durability. In *Proc. of IPTPS*, 2006.

[40] 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.

[41] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, 2002.

[42] J. Xiao, Z. Xiong, S. Wu, Y. Yi, H. Jin, and K. Hu. Disk Failure Prediction in Data Centers via Online Learning. In *Proc. of ICPP*, 2018.

[43] Y. Xu, K. Sui, R. Yao, H. Zhang, Q. Lin, Y. Dang, P. Li, K. Jiang, W. Zhang, J.-G. Lou, et al. Improving Service Availability of Cloud Systems by Predicting Disk Error. In *Proc. of USENIX ATC*, 2018.

[44] S. Zhang, Y. Liu, W. Meng, Z. Luo, J. Bu, S. Yang, P. Liang, D. Pei, J. Xu, Y. Zhang, Y. Chen, H. Dong, X. Qu, and L. Song. PreFix: Switch Failure Prediction in Datacenter Networks. In *Proc. of ACM SIGMETRICS*, 2018.

[45] B. Zhu, G. Wang, X. Liu, D. Hu, S. Lin, and J. Ma. Proactive Drive Failure Prediction for Large Scale Storage Systems. In *Proc. of IEEE MSST*, 2013.