Fast Proactive Repair in Erasure-Coded Storage: Analysis, Design, and Implementation

Xiaolu Li, Keyun Cheng, Zhirong Shen, and Patrick P. C. Lee

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 proactive 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 for both Reed-Solomon codes and Azure's Local Reconstruction Codes via mathematical analysis, large-scale simulation, and Amazon EC2 experiments.

1 Introduction

Failures are prevalent in large-scale storage clusters and manifest at disks or various storage components [9], [17], [34], [40]. Field studies report that disk replacements in production are more frequent than estimated by vendors [34], [40], and latent sector errors are commonly found in modern disks [3]. 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 [48]. Today's large-scale storage clusters increasingly adopt erasure coding to provide low-cost fault-tolerant storage (e.g., [1], [9], [15], [30], [32]), and reportedly save petabytes of storage compared to replication [15], [30].

While being storage-efficient, erasure coding incurs high repair penalty. As an example, we consider Reed-Solomon (RS) codes [38], which are a popular erasure coding construction used in production [1], [9], [30], [32]. 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 [7]. 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., regenerating codes [7] and Locally Repairable Codes (LRCs) [15], [39]), or designing repair-efficient techniques that apply to all practical erasure codes including RS codes (e.g., [4], [23], [24], [28], [43], [44]). 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 [5], [11], [21], [25], [27], [49], [51], [52], [54]; in some cases, the prediction accuracy can reach at least 95% [5], [21], [27], [52], [54], while having a very small false alarm rate (e.g., as low as 0.2-0.4% [52]). 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

- *An earlier conference version of this paper appeared at the 49th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'19) [41]. In this extended version, we extend FastPR to support Azure's Local Reconstruction Codes (Azure-LRC). We propose weighted repair scheduling and include new analysis and experiments for Azure-LRC.*
- *Xiaolu Li is with the School of Computer Science and Technology, Huazhong University of Science and Technology (E-mail: lixl666@hust.edu.cn). This work was partially done when Xiaolu Li worked as a Postdoctoral Fellow at The Chinese University of Hong Kong.*
- *Keyun Cheng, and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (E-mails: {kycheng, pclee}@cse.cuhk.edu.hk).*
- *Zhirong Shen is with the School of Informatics, Xiamen University, China (E-mail: shenzr@xmu.edu.cn).*
- *Corresponding author: Zhirong Shen.*

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**roactive **R**epair approach[1] 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. In addition, we study RS codes and Azure's Local Reconstruction Codes (Azure-LRC) [15]. While the repair traffic for RS codes is k chunks as discussed above, the repair traffic for Azure-LRC varies for different types of chunks. Thus, FastPR needs to address the subtle differences of the repair designs in RS codes and Azure-LRC. To this end, we present an in-depth study of FastPR through mathematical analysis, large-scale simulation, and Amazon EC2 experiments, and make the following contributions:

- We present mathematical analysis for RS codes and Azure-LRC on the optimal proactive repair in minimizing the total repair time. For example, in scattered repair, we show that the optimal proactive repair can reduce the repair time by up to 33.1% and 33.3% compared to the conventional reactive repair for RS codes and Azure-LRC, respectively. We further analyze the trade-off of the optimal proactive repair across the repair time, the amount of repair traffic, and the bandwidth resource usage.
- 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. In particular, we design a weighted repair scheduling algorithm to address the repair traffic differences for different types of chunks in Azure-LRC.
- We implement a FastPR prototype in C++. FastPR can run as a standalone system or be integrated with existing distributed storage systems. As a proof of concept, we integrate FastPR to HDFS of Hadoop 3.1.1 [2] without changing the HDFS codebase.
- We evaluate FastPR on Amazon EC2 with 25 instances. FastPR significantly reduces the repair times of both migration-only and reconstruction-only approaches for RS codes and Azure-LRC. For example, FastPR reduces the repair times of both migration-only and reconstruction-only approaches by up to 42.6% and 71.7%, respectively for different parameters of (n, k) in scattered repair for RS codes. For Azure-LRC, FastPR also reduces the repair times of both migration-only and reconstruction-only approaches by up to 64.5% and 58.5%, respectively, in scattered repair.

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

1. In our conference version [41], FastPR is an acronym of "Fast Predictive Repair". We now replace "predictive" by "proactive" to emphasize the property that FastPR proactively repairs an STF node before the STF node actually fails.

2 Background and Problem

2.1 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 [38], which are a well-known erasure code construction and have been widely deployed in production [1], [9], [30], [32], and Azure-LRC [15], a representative LRC construction deployed in Windows Azure Storage.

RS codes: 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. In RS codes, 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 our 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 the amount of lost data.

Azure-LRC: LRCs trade slightly higher redundancy for improved repair performance. There are different LRC constructions [19], and this work focuses on Azure-LRC [15]. An Azure-LRC(n, k, r) divides k data chunks into $\frac{k}{r}$ groups with r data chunks each; for the simplicity of our discussion, we assume that k is divisible by r in this paper. For each group, we compute a *local parity chunk* by the linear combination of the chunks within the same group. We also compute

$n-k-\frac{k}{r}$ *global parity chunks* by the linear combination of all k data chunks. For example, Azure-LRC$(10, 6, 3)$ includes six data chunks, two local parity chunks, and two global parity chunks. For an Azure-LRC, each lost data chunk or local parity chunk can be repaired locally by retrieving $\frac{k}{r}$ available chunks in the same group, while the repair of each lost global parity chunk still needs to retrieve k available data chunks or global parity chunks. Thus, LRC incurs an average amount of repair traffic less than k times the amount of the lost data and provides better repair performance than RS codes. As we show later, our analysis methodology can also be generalized for other LRC constructions [19].

2.2 Proactive Repair

Existing erasure codes (including RS codes, LRCs 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* repair approach that repairs the chunks stored in a soon-to-fail (STF) node 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 [16]. For example, RAIDShield [26] 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 [10], [34] (e.g., over half of the failed disks do not show SMART errors [34]). Nevertheless, machine learning has recently been shown to effectively predict disk failures in production environments based on SMART data [5], [21], [27], [49], [54] and additional system events [51]. 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 [20], [53].

Repair methods: We design the proactive repair mechanism by coupling two methods, namely *migration* and *reconstruction*, to repair the chunks of an STF node (e.g., determined by some failure prediction algorithms). 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.

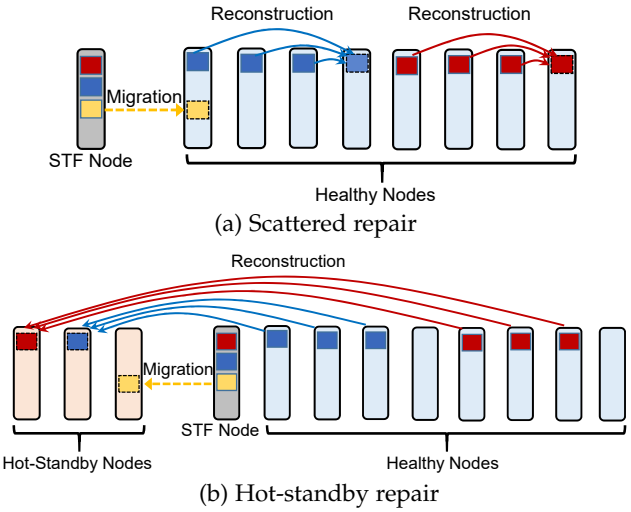


Fig. 1: Repair scenarios: (a) scattered repair and (b) hot-standby repair. Chunks of the same color belong to the same stripe. Different stripes are spread across the storage cluster.

Goal and assumptions: Our idea is to take advantage of both migration and reconstruction to maximize the proactive 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 [40].

Our work makes the following assumptions:

- 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 [37]) as opposed to multi-node repair [15], [37]. Nevertheless, if multiple failed nodes occur within a stripe, we can resort to the conventional reactive repair.
- 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 [40].
- 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.

2.3 Repair Scenarios

We study how we apply proactive 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 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.

3 Mathematical Analysis

We conduct simple mathematical analysis for RS codes and Azure-LRC to provide preliminary insights into the performance gain of the optimal proactive repair over the conventional reactive repair in a large-scale storage cluster. Note that our analysis is deterministic and the performance is based on the parameter values (i.e., no stochastic nature).

3.1 Analysis for RS Codes

In this subsection, we present analysis for RS codes.

General formulation: We first provide a general formulation that addresses the performance of both reactive and proactive 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 $T(x)$ and $B(x)$ be the total repair time and total repair traffic of proactive repair, respectively, and c be the chunk size. As both migration and reconstruction are performed in parallel, we have:

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

Also, the total repair traffic comprises both the traffic incurred by migration (i.e., $x \cdot c$) and the traffic incurred by reconstruction, (i.e., $(U-x) \cdot k \cdot c$). Thus, we have:

$$B(x) = x \cdot c + (U-x) \cdot k \cdot c. \quad (2)$$

	Repair time	Repair traffic
Proactive repair	$\frac{U \cdot t_r \cdot t_m}{G \cdot t_m + t_r}$	$\frac{U \cdot c \cdot (t_r + G \cdot t_m \cdot k)}{G \cdot t_m + t_r}$
Reactive repair	$\frac{U \cdot t_r}{G}$	$U \cdot k \cdot c$
Migration-only repair	$U \cdot t_m$	$U \cdot c$

TABLE 1: Total repair time and total repair traffic for proactive repair, reactive repair, and migration-only repair.

We can readily show that $T(x)$ is minimized when $x \cdot t_m = \frac{U-x}{G} \cdot t_r$, or equivalently, $x = \frac{U \cdot t_r}{G \cdot t_m + t_r}$. Table 1 shows the minimum proactive repair time and the corresponding repair traffic for proactive repair. In addition, Table 1 includes the repair time and repair traffic for reactive repair and migration-only repair. Reactive repair simply follows reconstruction without migration, so its repair time and repair traffic are obtained by setting $x = 0$ in Equations (1) and (2). Migration-only repair performs migration without reconstruction, so its repair time and repair traffic are obtained by setting $x = U$ in Equations (1) and (2).

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. We calculate the read time, transmission time, and write time for each repaired chunk.

To simplify our modeling, we make the following assumptions. 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 [18].

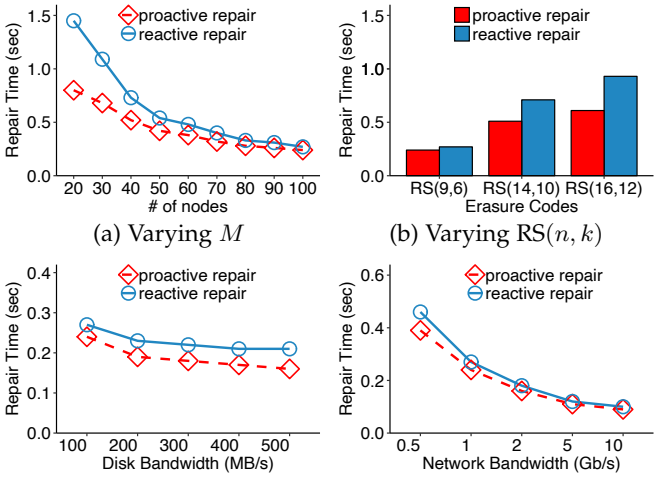
We first model t_m in migration, which applies to both scattered and hot-standby repairs. The read time, transmission time, and write time for each repaired chunk are c/b_d, c/b_n, and c/b_d, respectively. Thus,

$$t_m = \frac{c}{b_d} + \frac{c}{b_n} + \frac{c}{b_d}. \quad (3)$$

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,

$$t_r = \frac{c}{b_d} + \frac{k \cdot c}{b_n} + \frac{c}{b_d} \text{ (for scattered repair).} \quad (4)$$

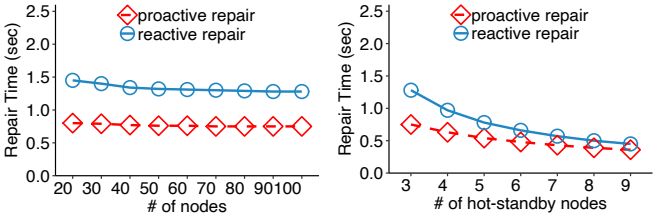
For the hot-standby repair, let h be the number of hot-standby nodes, such that $h \ll G$; hence, the transmissions and writes to the hot-standby nodes are the bottlenecks.



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

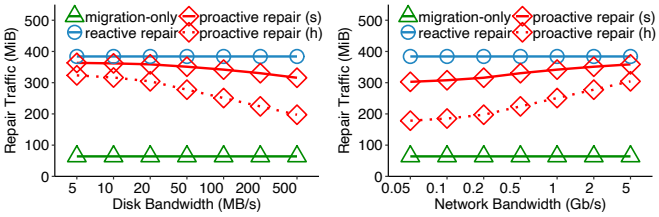
(c) Varying b_d (d) Varying b_n

Fig. 2: Repair time analysis in scattered repair.



(a) Varying M (b) Varying h

Fig. 3: Repair time analysis in hot-standby repair.



(a) Varying b_d (b) Varying b_n

Fig. 4: Repair traffic analysis; "s" refers to scattered repair, while "h" refers to hot-standby repair.

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,

$$t_r = \frac{c}{b_d} + \frac{G \cdot k \cdot c}{h \cdot b_n} + \frac{G \cdot c}{h \cdot b_d} \quad \text{(for hot-standby repair).} \quad (5)$$

Repair time analysis: We first study the performance gain of the optimal proactive repair over the conventional reactive repair in terms of the repair time (Table 1) via mathematical analysis. 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; note that it is not always the case that we can repair $G = \frac{M-1}{k}$ chunks in a repair round, and we relax this assumption in Section 4.

We consider the following default configurations. We set $M = 100$, $U = 1{,}000$ chunks of size $c = 64\,\text{MB}$ each, $b_d = 100\,\text{MB/s}$, and $b_n = 1\,\text{Gb/s}$. We consider RS$(9, 6)$, the default erasure coding configuration in QFS [32]. 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.

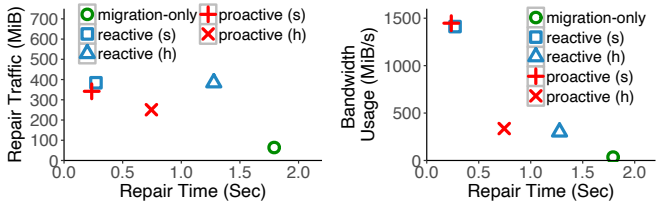
Figure 2 first shows the repair time in scattered repair. Proactive repair shows a higher performance gain than reactive repair when the number of nodes is small (Figure 2(a)), k is large (Figure 2(b)), b_d is large (Figure 2(c)), and b_n is small (Figure 2(d)). The reason is that the repair penalty due to the amplified repair traffic in reactive repair becomes more significant in such cases. Overall, proactive 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, proactive repair is more significant (Figure 3(b)). For example, when $h = 3$, proactive repair reduces the repair time by 41.3%.

Repair traffic analysis: We analyze the repair traffic of different repair methods in Table 1 based on our default configurations. In our analysis, we measure the repair traffic per chunk. Figure 4 shows the results. When b_d is limited (Figure 4(a)), proactive repair schedules more chunks to be repaired by reconstruction to exploit the benefit of repairing chunks in parallel, and hence incurs more repair traffic. When b_n is limited (Figure 4(b)), proactive repair schedules more chunks to be repaired by migration to reduce repair traffic. Overall, migration-only repair incurs the minimum repair traffic but has the longest repair time, while proactive repair reduces the repair traffic by up to 17.8% and 48.6% compared to reactive repair in scattered repair and hot-standby repair, respectively.

Trade-off analysis: We analyze the trade-off between the repair time and repair traffic for different repair methods in Table 1. We report the average repair time and average repair traffic per chunk based on the default configurations (i.e., $M = 100$, $U = 1{,}000$ chunks of size $c = 64\,\text{MB}$ each, $b_d = 100\,\text{MB/s}$, and $b_n = 1\,\text{Gb/s}$). Figure 5(a) shows the results. Migration-only repair incurs the least repair traffic, but requires the longest repair time as it is bottlenecked by the available bandwidth of the STF node. In scattered repair, proactive repair has 13.2% less repair time and 11.0% less repair traffic than reactive repair. In hot-standby repair, proactive repair has 41.7% less repair time and 34.7% less repair traffic than reactive repair. Overall, proactive repair offers better trade-off between repair time and repair traffic in both repair scenarios.

We also study the trade-off between repair time and *bandwidth usage* (defined by the amount of repair traffic divided by the repair time) in Figure 5(b), which shows that proactive repair offers better trade-off between repair time and average bandwidth usage in both scenarios. Although migration-only repair has the minimum bandwidth usage, it is bottlenecked by the bandwidth of the STF node and has the longest repair time. Both proactive repair and reactive repair benefit from repairing chunks in parallel at different storage nodes, while proactive repair uses slightly more bandwidth for more repair time reduction. In scattered repair, proactive repair uses only 2.6% more bandwidth than reactive repair,



(a) Repair Traffic (b) Bandwidth Usage

Fig. 5: Trade-off analysis; "s" refers to scattered repair, while "h" refers to hot-standby repair.

while it reduces repair time by 13.2%. In hot-standby repair, proactive repair has 11.8% more bandwidth usage, while reducing the repair time by 41.7%.

Discussion: Our analysis currently only focuses on the trade-off across the repair time, repair traffic, and bandwidth usage, but does not consider other types of system resources such as CPU overhead, memory usage, and disk I/O overhead. A more comprehensive analysis on the performance impact of different repair methods on resource consumption is posed as a future work.

3.2 Analysis for Azure-LRC

In this subsection, we present the analysis for Azure-LRC. In Azure-LRC, the chunks in the STF node comprises data chunks, local parity chunks, and global parity chunks. Unlike RS codes, the repair traffic of Azure-LRC under reconstruction varies for different types of chunks. For example, for Azure-LRC$(10, 6, 3)$, the repair of a data chunk or a local parity chunk retrieves three available chunks, while the repair of a global parity chunk still requires retrieving six surviving chunks. Thus, our formulation should address the repair traffic difference in reconstruction for Azure-LRC.

Formulation of reactive repair: We divide the chunks in the STF node into two groups, namely the *local parity group* and the *global parity group*. The local parity group corresponds to the data chunks and the local parity chunks, and the repair of a chunk in the local parity group retrieves k_l chunks. In contrast, the global parity group corresponds to the global parity chunks, and the repair of a chunk in the global parity group retrieves k_g chunks. For example, for Azure-LRC$(10, 6, 3)$, $k_l = 3$ and $k_g = 6$. Note that our analysis can be generalized for other LRC constructions [19], by setting the proper k_l and k_g.

We divide the reconstruction process into multiple rounds. In each round, we repair chunks from either the local parity group or the global parity group (but not both)[2]. Let U_l be the total number of chunks in the local parity group in the STF node. Thus, we can repair $G_l = \frac{M-1}{k_l}$ chunks from the local parity group in a round, where M is the total number of storage nodes. Let $t_{r,l}$ be the time to repair a chunk in the local parity group. Then the time to repair all chunks in the local parity group is $\frac{U_l \cdot t_{r,l}}{G_l}$. Similarly, let U_g be the total number of chunks in the global parity group in the STF node, and $t_{r,g}$ be the repair time of a chunk in the global parity group. Then we can repair $G_g = \frac{M-1}{k_g}$ chunks from the global parity group in a round, and the total repair time of all chunks in the global parity group is $\frac{U_g \cdot t_{r,g}}{G_g}$ Thus, the total reconstruction time (denoted by T_R) is

$$T_R = \frac{U_l \cdot t_{r,l}}{G_l} + \frac{U_g \cdot t_{r,g}}{G_g}. \tag{6}$$

Formulation of proactive repair: Let x_l and x_g be the numbers of chunks selected for migration from a local parity group and a global parity group. Let t_m be the time to migrate a chunk. Thus, the time for migration (denoted by $T_m(x_l, x_g)$) is

$$T_m(x_l, x_g) = (x_l + x_g) \cdot t_m. \tag{7}$$

We can show that the numbers of chunks repaired by reconstruction in the local parity group and the global parity group are $U_l - x_l$ and $U_g - x_g$, respectively. Thus, the time spent for reconstruction (denoted by $T_r(x_l, x_g)$) is

$$T_r(x_l, x_g) = \frac{(U_l - x_l) \cdot t_{r,l}}{G_l} + \frac{(U_g - x_g) \cdot t_{r,g}}{G_g}. \tag{8}$$

Let $T(x_l, x_g)$ be the total repair time of proactive repair. Then we have

$$T(x_l, x_g) = \max(T_m(x_l, x_g), T_r(x_l, x_g)). \tag{9}$$

When $T_m(x_l, x_g) = T_r(x_l, x_g)$, the total repair time can be minimized. Thus, we can obtain the optimal proactive repair time by solving the following optimization problem:

$$\begin{aligned} \min_{x_l, x_g} \quad & (x_l + x_g) \cdot t_m \\ \text{subject to} \quad & T_m(x_l, x_g) = T_r(x_l, x_g), \\ & 0 \le x_l \le U_l, \\ & 0 \le x_g \le U_g. \end{aligned} \tag{10}$$

Modeling repair scenarios: We model the repair time of a chunk in the local parity group $t_{r,l}$, as well as the repair time of a chunk in the global parity group $t_{r,g}$, in both scattered repair and hot-standby repair following the same method in Section 3.1. Note that the migration time is shown in Equation (3).

Let b_d and b_n be the disk and network bandwidths, respectively, c be the chunk size, and h be the number hot-standby nodes. We replace k by k_l and G by G_l in Equations (4) and (5). We obtain $t_{r,l}$ in scattered repair and hot-standby repair as follows:

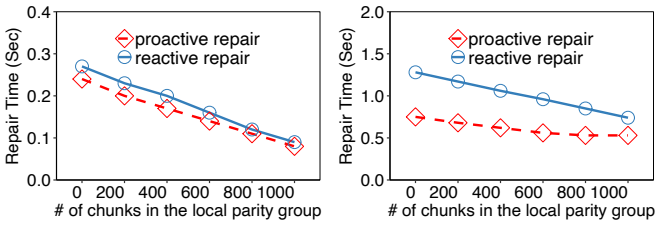
$$t_{r,l} = \begin{cases} \frac{c}{b_d} + \frac{k_l \cdot c}{b_n} + \frac{c}{b_d} & \text{(scattered repair)}, \\ \frac{c}{b_d} + \frac{G_l \cdot k_l \cdot c}{h \cdot b_n} + \frac{G_l \cdot c}{h \cdot b_d} & \text{(hot-standby repair)}. \end{cases} \tag{11}$$

Similarly, we can obtain $t_{r,g}$ for both repair approaches:

$$t_{r,g} = \begin{cases} \frac{c}{b_d} + \frac{k_g \cdot c}{b_n} + \frac{c}{b_d} & \text{(scattered repair)}, \\ \frac{c}{b_d} + \frac{G_g \cdot k_g \cdot c}{h \cdot b_n} + \frac{G_g \cdot c}{h \cdot b_d} & \text{(hot-standby repair)}. \end{cases} \tag{12}$$

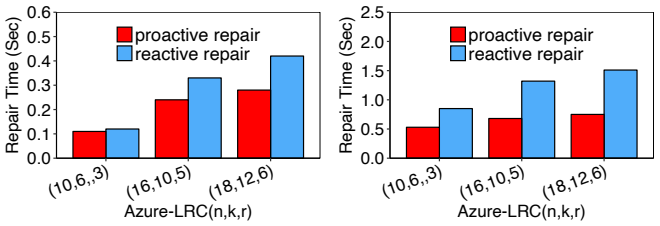
Analysis: We study the performance gain of the proactive repair over the reactive repair for Azure-LRC, based on the

2. Our digital supplementary file compares two reactive repair methods: the *independent repair* that only repair chunks from either the local parity group or the global parity group in a round, and the *mixed repair* that can repair chunks from both the local parity group and the global parity group in a round. We show that the independent repair outperforms the mixed repair, and hence we choose the independent repair as our reactive repair method.



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

Fig. 6: Mathematical analysis for different chunk distributions in the STF node for Azure-LRC(10,6,3).



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

Fig. 7: Mathematical analysis for different Azure-LRC configurations.

default parameters in the analysis of Section 3.1. In addition, we fix $U_l + U_g = 1000$. We calculate the reactive repair time by Equation (6) and the optimal proactive repair time by solving the optimization problem of Equation (10). We show the repair time per chunk in our analysis.

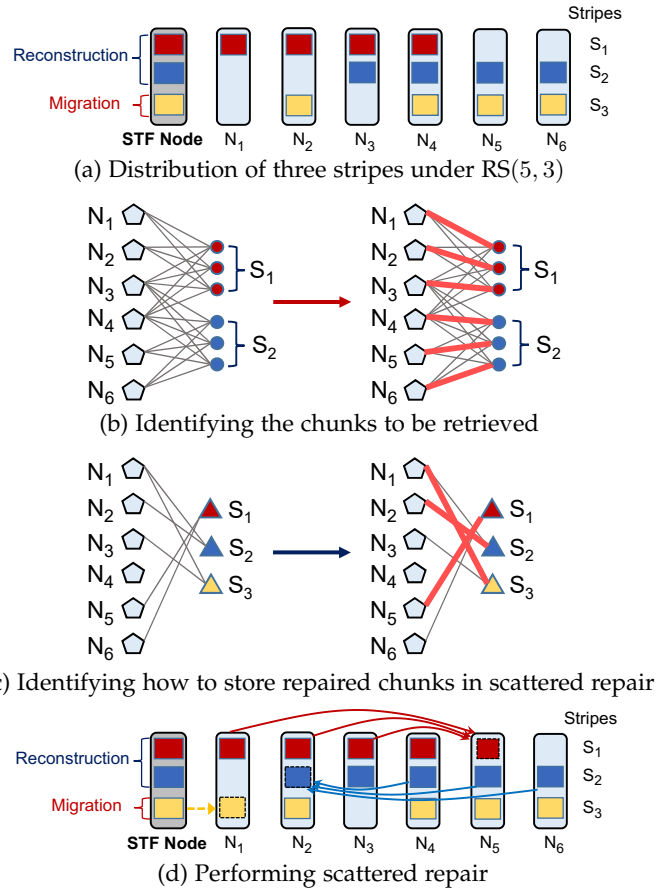
We first study the impact of the chunk distribution in the STF node by varying U_l from 0 to 1000. We focus on Azure-LRC(10,6,3). Figure 6 shows that the repair time decreases as the number of chunks in the local parity group increases, since the repair traffic of a chunk in the local parity group is less than that in the global parity group. The optimal proactive repair outperforms the reactive repair in both scattered repair and hot-standby repair. The performance gain of the optimal proactive repair over the reactive repair is up to 15% when $U_l = 400$ in scattered repair, and up to 41.8% when $U_l = 200$ in hot-standby repair.

We next analyze the performance gain of proactive repair for different Azure-LRC configurations. We set $U_l = 1000 \times (k + \frac{k}{r})/n$ for Azure-LRC(n, k, r). Figure 7 shows that the maximum performance gains obtained by the optimal proactive repair over the reactive repair for Azure-LRC are 33.3% and 50.3% for scattered repair and hot-standby repair, respectively.

4 Fast Proactive Repair

We present the design of FastPR, which couples both migration and reconstruction to achieve fast proactive repair. FastPR aims to identify, with polynomial complexity, a repair solution that minimizes the repair time (Section 3). Note that FastPR itself does not perform failure prediction; instead, it builds on existing failure prediction mechanisms (e.g., [5], [21], [27], [49], [51], [54]) and provides a fast repair solution for an STF node before the STF node actually fails.

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



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

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.

In the following discussion, we focus on RS codes. In Section 4.4, we extend the design to Azure-LRC.

4.1 Design Overview

We first provide an overview of how FastPR performs both migration and reconstruction in a repair round. Figure 8 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 8(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.

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 8(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 8(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 8(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 [6].

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 8(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 [6], 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 8(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 8(d) for scattered repair). In the following, we show how we identify the chunks for migration and reconstruction in each repair round.

4.2 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 3), 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 4.1 and design a greedy algorithm to find a subset of chunks of the STF node that can be reconstructed in parallel. Based on the reconstruction sets, we then schedule the chunks to be repaired through either migration or reconstruction (Section 4.3).

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 9 shows an example of finding all reconstruction 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 9(a)). We fur-

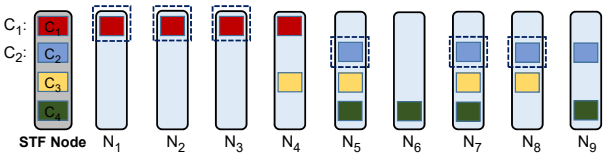
Algorithm 1 Finding Reconstruction Sets

```
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
```

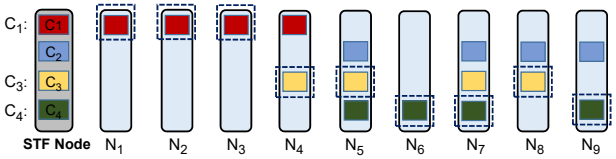
ther optimize $\mathcal{R}$ by checking if it can include more chunks in $\mathcal{C}$. By replacing C_2 in $\mathcal{R}$ with C_3 in $\mathcal{C}$, we see that C_4 can now be added to the reconstruction set, so the new reconstruction set becomes $\mathcal{R} = \{C_1, C_3, C_4\}$ (Figure 9(b)). The remaining C_2 forms another reconstruction set. Thus, we have two reconstruction sets $\{\mathcal{R}_1, \mathcal{R}_2\} = \{\{C_1, C_3, C_4\}, \{C_2\}\}$.

4.3 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,



(a) An initial reconstruction set



(b) An optimized reconstruction set

Fig. 9: 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.

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 3). For t_m, we can derive it via Equation (3). For t_r, we derive it via Equations (4) and (5) 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 $\ell = 1$ 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 $c_r = |\mathcal{R}_\ell|$ (Line 4). If $|\mathcal{R}_{\ell+1} \cup \cdots \cup \mathcal{R}_u| \leq c_m$, it implies that $\mathcal{R}_{\ell+1} \cup \cdots \cup \mathcal{R}_u$ (denoted by $\mathcal{M}_\ell$) can be repaired through migration, in parallel with the reconstruction of $\mathcal{R}_\ell$. 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 $\mathcal{M}_\ell = \mathcal{R}'_x \cup \mathcal{R}_{x+1} \cup \cdots \cup \mathcal{R}_u$, which are the c_m chunks to be repaired through migration (Line 12), in parallel with the reconstruction of $\mathcal{R}_\ell$ in the same repair round. We update ℓ and u (Lines 13-14) and iterate for another repair round.

Example: Figure 10 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

Algorithm 2 Repair Scheduling

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

Sort $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_d\}$, where $|\mathcal{R}_1| \geq |\mathcal{R}_2| \geq \cdots \geq |\mathcal{R}_d|$
Initialize $\ell = 1$ and $u = d$
while true do
  Compute c_m from c_r, where $c_r = |\mathcal{R}_\ell|$
  if $|\mathcal{R}_{\ell+1} \cup \cdots \cup \mathcal{R}_u| \leq c_m$ **then**
    $\mathcal{M}_\ell = \mathcal{R}_{\ell+1} \cup \cdots \cup \mathcal{R}_u$
    break
  end if
  Find the largest x, where $\sum_{i=x}^{u} |\mathcal{R}_i| > c_m$
  Find a subset $\mathcal{R}'_x \subset \mathcal{R}_x$, where $|\mathcal{R}'_x| + \sum_{i=x+1}^{u} |\mathcal{R}_i| = c_m$
  Set $\mathcal{R}_x = \mathcal{R}_x - \mathcal{R}'_x$
  Set $\mathcal{M}_\ell = \mathcal{R}'_x \cup \mathcal{R}_{x+1} \cup \cdots \cup \mathcal{R}_u$
  Set $\ell = \ell + 1$
  Set $u = x$
end while

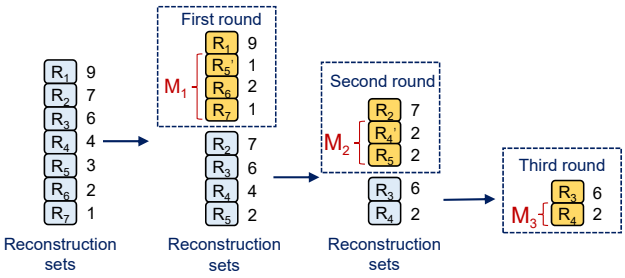


Fig. 10: Example of repair scheduling. We show the number of remaining chunks to be repaired in each reconstruction set.

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.

4.4 Extensions for Azure-LRC

Recall that the reconstruction of a lost chunk in Azure-LRC retrieves a different number of available chunks, depending on whether the lost chunk belongs to the local parity group or the global parity group; in contrast, the reconstruction of a lost chunk in RS codes always retrieves k chunks. In this subsection, we extend FastPR for Azure-LRC, by proposing *weighted repair scheduling* to account for the repair traffic difference in the reconstruction process of Azure-LRC.

Design idea: We define two types of reconstruction sets: *local reconstruction sets* and *global reconstruction sets*, which contain the chunks of the STF node belonging to the local parity group and the global parity group, respectively. Each type of reconstruction sets can be independently found by Algorithm 1.

We also introduce a new *weight* metric (denoted by $w_\mathcal{R}$) to quantify the repair efficiency of a local/global reconstruction set $\mathcal{R}$. To calculate $w_\mathcal{R}$, we derive $t_{r,l}$ and $t_{r,g}$ (i.e., the times to repair a chunk of the local and global parity groups through reconstruction, respectively) from Equations (11) and (12), respectively. We also derive t_m (i.e., the time to repair a chunk through migration) via Equation (3). Then we calculate the maximum number of chunks that can be migrated in a repair round (denoted by c_m) as $c_m = \frac{t_{r,l}}{t_m}$ (or $c_m = \frac{t_{r,g}}{t_m}$) for repairing a chunk in the local (or global) parity group. Finally, we calculate $w_\mathcal{R} = \frac{|\mathcal{R}|+c_m}{t_{r,l}}$, where $|\mathcal{R}|$ is the number of chunks being repaired by reconstruction in $\mathcal{R}$.

Intuitively, the physical meaning of $w_\mathcal{R}$ represents the rate of repairing the chunks of the STF node in a repair round. In weighted repair scheduling, we schedule the repair by first selecting a reconstruction set with a higher $w_\mathcal{R}$.

Algorithm 3 Weighted Repair Scheduling

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

1: **for** each reconstruction set $\mathcal{R}$ **do**
2: **if** $\mathcal{R}$ is a local reconstruction set **then**
3: $c_m = t_{r,l}/t_m$
4: $w_\mathcal{R} = (|\mathcal{R}| + c_m)/t_{r,l}$
5: **else**
6: $c_m = t_{r,g}/t_m$
7: $w_\mathcal{R} = (|\mathcal{R}| + c_m)/t_{r,g}$
8: **end if**
9: **end for**
10: Sort $\{\mathcal{R}_1, \mathcal{R}_2, \cdots, \mathcal{R}_d\}$, where $w_{\mathcal{R}_1} \geq w_{\mathcal{R}_2} \geq \cdots \geq w_{\mathcal{R}_d}$
11: Initialize $\ell = 1$ and $u = d$
12: **while true do**
13: **if** $\mathcal{R}_\ell$ is a local reconstruction set **then**
14: $c_m = t_{r,l}/t_m$
15: **else**
16: $c_m = t_{r,g}/t_m$
17: **end if**
18: **if** $|\mathcal{R}_{\ell+1} \cup \cdots \cup \mathcal{R}_u| \leq c_m$ **then**
19: $\mathcal{M}_\ell = \mathcal{R}_{\ell+1} \cup \cdots \cup \mathcal{R}_u$
20: **break**
21: **end if**
22: Find the largest x, where $\sum_{i=x}^{u} |\mathcal{R}_i| > c_m$
23: Find a subset $\mathcal{R}'_x \subset \mathcal{R}_x$, where $|\mathcal{R}'_x| + \sum_{i=x+1}^{u} |\mathcal{R}_i| = c_m$
24: Set $\mathcal{M}_\ell = \mathcal{R}'_x \cup \mathcal{R}_{x+1} \cup \cdots \cup \mathcal{R}_u$
25: Set $\mathcal{R}_x = \mathcal{R}_x - \mathcal{R}'_x$
26: Set $\ell = \ell + 1$
27: Set $u = x$
28: **end while**

Algorithm details: Algorithm 3 presents the pseudo-code of our weighted repair scheduling algorithm. We first calculate the weight for each local/global reconstruction set (Lines 1-9). Then we sort all d reconstruction sets by their weights in monotonically descending order (d is the total number of reconstruction sets) (Line 10). We use the indices ℓ and u to refer to the currently considered reconstruction sets with the largest and smallest weights, respectively, and we initialize $\ell = 1$ and $u = d$. In each repair round, we choose $\mathcal{R}_\ell$ for reconstruction. Then we compute the maximum number of chunks that can be migrated (i.e., c_m) in the repair round (Lines 13-17) and select the reconstruction sets that can be repaired through migration (Lines 18-27); note that the workflow is similar to that in Algorithm 2.

Example: Figure 11 shows an example of weighted repair scheduling. Suppose that we have six local reconstruction sets (i.e., $\mathcal{R}_{l,1}, \mathcal{R}_{l,2}, \mathcal{R}_{l,3}, \mathcal{R}_{l,4}, \mathcal{R}_{l,5}, \mathcal{R}_{l,6}$) and four global reconstruction sets (i.e., $\mathcal{R}_{g,1}, \mathcal{R}_{g,2}, \mathcal{R}_{g,3}, \mathcal{R}_{g,4}$). We let $t_{r,l} = 3$,

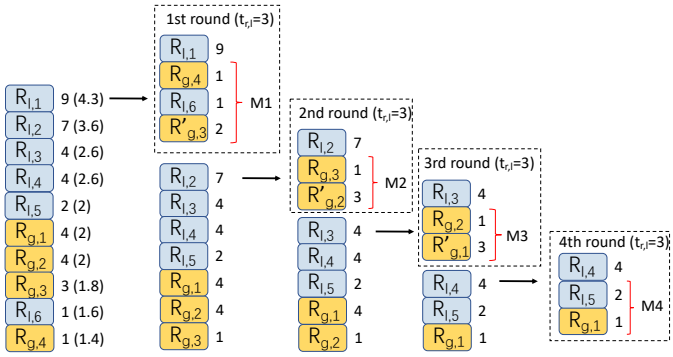


Fig. 11: Example of weighted repair scheduling for Azure-LRC. We show the number of remaining chunks to be repaired in each reconstruction set. Each number in the brackets represents the weight of the corresponding reconstruction set.

$t_{r,g} = 5$, and $t_m = 0.75$. We calculate the weight for each reconstruction set and sort the reconstruction sets by their weights. For example, for $\mathcal{R}_{l,1}$, suppose that $|\mathcal{R}_{l,1}| = 9$. Then we have $c_m = \frac{t_{r,l}}{t_m} = 4$, and hence the weight of $\mathcal{R}_{l,1}$ is 4.3. In the first repair round, we choose $\mathcal{R}_{l,1}$, which has the largest weight, to be repaired through reconstruction. Since we can migrate at most $c_m = 4$ chunks in the reconstruction of $\mathcal{R}_{l,1}$, we select one chunk from $\mathcal{R}_{g,4}$, one chunk from $\mathcal{R}_{l,6}$, and two chunks from $\mathcal{R}_{g,3}$ to be repaired through migration. The scheduling for other repair rounds follows the similar procedures.

5 Implementation

We have built a FastPR prototype in C++, including the coding operations for chunk reconstruction using Jerasure v1.2 [35]. Our FastPR prototype supports both RS codes and Azure-LRC. Our prototype has around 4,300 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 for RS codes: 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 [2], which supports RS codes by design. Figure 12 illustrates the integration of FastPR with HDFS under RS codes. Specifically, HDFS comprises a NameNode

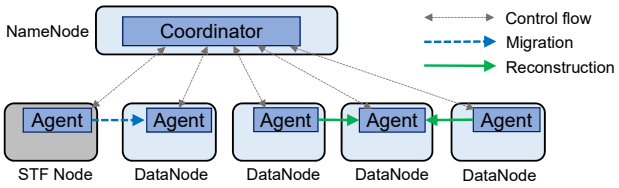


Fig. 12: Integration of FastPR with HDFS under RS codes.

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.

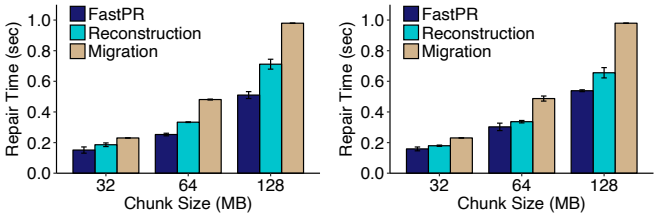
Standalone system: Since Hadoop-3.1.1 only supports RS codes but not Azure-LRC, we also implement FastPR as a standalone system for Azure-LRC, such that the coordinator and all agents run as a daemon in a distinct server and each agent uses the local disk for storage.

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 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.

6 Performance Evaluation

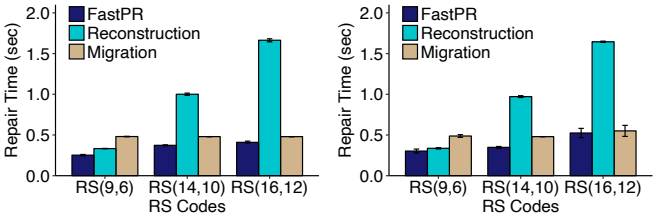
We conduct experiments on Amazon EC2 to understand the performance of FastPR for RS codes (Section 6.1) and Azure-LRC (Section 6.2). In our digital supplementary file, we present additional evaluation results on Amazon EC2 and evaluation results on the large-scale simulation.

In our Amazon EC2 experiments, we compare FastPR with two approaches: (i) *migration-only*, in which we directly migrate all the chunks of the STF node to other healthy nodes, and (ii) *reconstruction-only*, in which we find the reconstruction sets based on Algorithm 1 and repair each of them in a repair round by reconstruction only (note that it corresponds to the conventional reactive repair). We plot the average repair time per chunk over five runs, with an error bar showing the 95% confidence interval based on the student's t-distribution (some error bars may be invisible due to very small deviations).



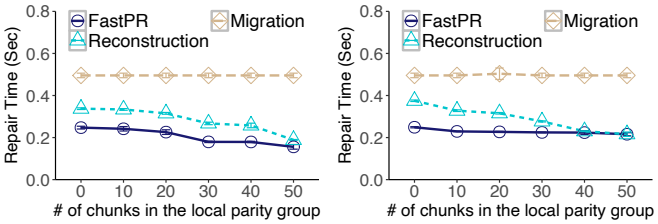
(a) Scattered repair (b) Hot-standby repair

Fig. 13: Experiment 1: Impact of the chunk size.



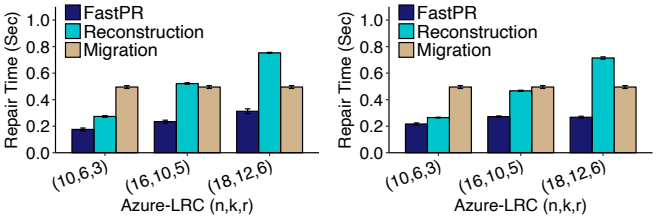
(a) Scattered repair (b) Hot-standby repair

Fig. 14: Experiment 2: Impact of different erasure codes.



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

Fig. 15: Experiment 3: Impact of chunk distribution in the STF node under Azure-LRC.



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

Fig. 16: Experiment 4: Impact of different Azure-LRC configurations.

6.1 Experiments for RS Codes

We evaluate FastPR for RS codes atop HDFS of Hadoop 3.1.1 [2] on Amazon EC2. 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 5), 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.

Experiment 1 (Impact of the chunk size): We first evaluate the impact of the chunk size, varied from 32 MB to 128 MB (where the packet size is fixed as 4 MB). Figure 13 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 2 (Impact of different erasure codes): We now evaluate the repair time per chunk for different erasure codes. We focus on RS$(9, 6)$, RS$(14, 10)$, and RS$(16, 12)$. Figure 14 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.

6.2 Experiments for Azure-LRC

We next evaluate FastPR for Azure-LRC, as a standalone system, on Amazon EC2. Our evaluation settings follow those in Section 6.1.

Experiment 3 (Impact of chunk distribution in the STF node under Azure-LRC): We first study the impact of chunk distribution in the STF node. We vary the number of chunks in the local parity group in the STF node from 0 to 50 (note that we fix the total number of chunks in the STF node to be 50).

Figure 15(a) shows the results for scattered repair. We observe that the repair time per chunk of both reconstruction-only and FastPR decreases when the number of chunks in the local parity group in the STF node increases, as the time to repair a chunk in the local parity group is less than that to repair a chunk in the global parity group. Our results show that FastPR reduces the repair times by up to 32.9% and 68.5% compared to reconstruction-only and migration-only, respectively.

Figure 15(b) shows the results for hot-standby repair. FastPR reduces the repair times of reconstruction-only and migration-only by up to 34.1% and 57.4%, respectively. Note that when the number of chunks in the local parity group is larger than 40, the repair time of FastPR is only slightly less than that of the reconstruction-only approach, as most chunks of the STF node can be locally repaired with significantly less repair traffic than in RS codes. The benefit of FastPR becomes marginal.

Experiment 4 (Impact of different Azure-LRC configura-

tions): We evaluate the repair time per chunk for different Azure-LRC configurations. Figure 16 shows the results. Overall, FastPR outperforms reconstruction-only and migration-only approaches. Specifically, compared to the reconstruction-only approach, FastPR reduces the repair times by up to 58.5% and 62.5% in scattered repair and hot-standby repair, respectively (for Azure-LRC$(18, 12, 6)$). Compared to the migration-only approach, FastPR reduces the repair times by up to 64.5% and 56.3% in scattered repair and hot-standby repair (for Azure-LRC$(10, 6, 3)$).

7 Related Work

Proactive fault tolerance: Our work builds on the potential of accurate failure prediction [5], [21], [27], [49], [51], [54], 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 replication [8], [45] injects redundant data before failures occur to avoid traffic bursts during repair when failures happen. RAIDShield [26] uses the reallocated sector count to identify STF disks and replaces them in advance. ProCode [22] 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 [7], [33], [36], [47] and LRCs [14], [15], [19], [39], [46]). FastPR proposes proactive repair, and it improves the performance of reactive repair for both RS codes and Azure-LRC.

Some studies design repair-efficient techniques that schedule the repair for existing erasure codes, such as lazy repair [4], [44], parallelizing partial repair operations for RS Codes to improve the repair performance [23], [24], [28], scheduling the repair of multiple stripes [42], [43], [50], and combining the locality properties of LRCs and topology [13]. In contrast, our FastPR takes a proactive repair approach to improve the repair performance.

To fully exploit bandwidth resources in repair, parity declustering [12], [29] distributes stripes across nodes to parallelize repair operations. RAMCloud [31] 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.

8 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 proactive 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. We demonstrate that FastPR works for both RS codes and Azure-LRC. Our mathematical analysis, large-scale simulation (see the digital supplementary file), and Amazon EC2 experiments demonstrate that FastPR outperforms the conventional reactive repair approach that triggers repair operations only after a failed node is detected. Our future work is to comprehensively analyze the resource consumption of proactive repair as discussed in Section 3.1.

References

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

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

[3] 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.

[4] 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.

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

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

[7] 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.

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

[9] 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.

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

[11] S. Han, P. P. Lee, Z. Shen, C. He, Y. Liu, and T. Huang. Toward adaptive disk failure prediction via stream mining. In *Proc. of IEEE ICDCS*, 2020.

[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] Y. Hu, L. Cheng, Q. Yao, P. P. Lee, W. Wang, and W. Chen. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proc. of USENIX FAST*, 2021.

[14] C. Huang, M. Chen, and J. Li. Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems. *ACM Trans. on Storage*, 9(21):1–28, 2013.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] O. Kolosov, G. Yadgar, M. Liram, I. Tamo, and A. Barg. On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes. In *Proc. of USENIX ATC*, 2018.

[20] 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.

[21] 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.

[22] 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.

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

[24] 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.

[25] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making disk failure predictions smarter! In *Proc. of USENIX FAST*, 2020.

[26] 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.

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

[28] 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.

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

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

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

[32] 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.

[33] 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.

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Z. Shen, X. Li, and P. P. C. Lee. Fast Predictive Repair in Erasure-Coded Storage. In *Proc. of IEEE/IFIP DSN*, 2019.

[42] Z. Shen, J. Shu, Z. Huang, and Y. Fu. ClusterSR: Cluster-Aware Scattered Repair in Erasure-Coded Storage. In *Proc. of IEEE IPDPS*, 2020.

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

[44] 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.

[45] 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.

[46] I. Tamo and A. Barg. A Family of Optimal Locally Recoverable Codes. *IEEE Trans. on Information Theory*, 2014.

[47] 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.

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

[49] 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.

[50] L. Xu, M. Lyu, Q. Li, L. Xie, and Y. Xu. SelectiveEC: Selective Reconstruction in Erasure-coded Storage Systems. In *Proc. of USENIX HotStorage*, 2020.

[51] 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.

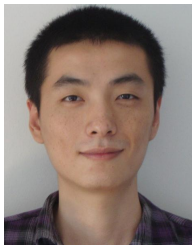
[52] J. Zhang, P. Huang, K. Zhou, M. Xie, and S. Schelter. HDDse: Enabling High-Dimensional Disk State Embedding for Generic Failure Detection System of Heterogeneous Disks in Large Data Centers. In *Proc. of USENIX ATC*, 2020.

[53] 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.

[54] 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.

Xiaolu Li received the B.Eng. degree from University of Science and Technology of China in 2016, and the Ph.D. degree in Computer Science and Engineering from The Chinese University of Hong Kong in 2020. She is now a lecturer at the School of Computer Science and Technology, Huazhong University of Science and Technology. Her current research interests include distributed storage system, erasure-coding, and container storage.

Keyun Cheng received the B.Eng. degree in Software Engineering from Sun Yat-Sen University in 2018, and the M.Sc. degree in Computer Science from The Chinese University of Hong Kong in 2019. He is now a Ph.D. student in Computer Science and Engineering at The Chinese University of Hong Kong. His research interests include distributed storage systems, and multi-cloud storage systems.



Zhirong Shen received the B.S. degree from University of Electronic Science and Technology of China in 2010, and the Ph.D. degree in Computer Science from Tsinghua University in 2016. He is now an associate professor at Xiamen University. His current research interests include storage reliability and storage security.

Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now a Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, and cloud computing.