Harnessing Parallelism for Fast Data Repair in MSR-Coded Storage

XIAOLU LI, HAN YUAN, XUAN LIU, and JUNLONG ZHANG, Huazhong University of Science and Technology, China
PATRICK P. C. LEE, The Chinese University of Hong Kong, China
YUCHONG HU and DAN FENG, Huazhong University of Science and Technology, China

Minimum-storage regenerating (MSR) codes are provably optimal erasure codes that minimize the repair bandwidth (i.e., the amount of traffic being transferred during a repair operation), while minimizing storage redundancy, in distributed storage systems. However, the practical repair performance of MSR codes still has significant room for improvements, as their mathematical structure makes repair operations difficult to parallelize. In this paper, we present HyperParaRC, a parallel repair framework for MSR codes. HyperParaRC leverages the sub-packetization nature of MSR codes to parallelize the repair of sub-blocks and balance repair load (i.e., the amount of traffic sent or received by a node) across available nodes. We first demonstrate that there exists a trade-off between repair bandwidth and maximum repair load. We then propose an affinity-based heuristic for HyperParaRC, which approximately minimizes the maximum repair load by examining the bandwidth incurred during sub-block computations and significantly reduces the search time for large coding parameters compared with our earlier work, ParaRC. Based on our affinity-based heuristic, we further design a full-node recovery mechanism for HyperParaRC that combines both intra-stripe and inter-stripe parallel repair scheduling to repair multiple lost blocks in a failed node. We prototype HyperParaRC on Hadoop HDFS and evaluate it on Alibaba Cloud. Our evaluation results show that HyperParaRC reduces both single-block repair and full-node recovery times compared with state-of-the-art repair approaches.

CCS Concepts: • **Information systems** → **Storage recovery strategies**; **Distributed storage**.

Additional Key Words and Phrases: Erasure coding, Distributed Storage Systems

ACM Reference Format:
Xiaolu Li, Han Yuan, Xuan Liu, Junlong Zhang, Patrick P. C. Lee, Yuchong Hu, and Dan Feng. 2024. Harnessing Parallelism for Fast Data Repair in MSR-Coded Storage. *ACM Trans. Storage* 1, 1, Article 1 (January 2024), 37 pages. https://doi.org/0000001.0000001

This work was supported in part by the National Natural Science Foundation of China (62302175 and 62272185), Research Grants Council of Hong Kong (AoE/P-404/18), and Research Matching Grant Scheme. Corresponding author: Patrick P. C. Lee (pclee@cse.cuhk.edu.hk).

An earlier version of this article appeared in [22]. In this extended version, we propose an affinity-based heuristic that quickly identifies an efficient repair solution with both low repair bandwidth and low minimum-maximum repair load under large coding parameters. We further design an intra-stripe and inter-stripe parallel repair scheduling algorithm for the full-node recovery of MSR codes based on our affinity-based heuristic. We implement our extended system, namely HyperParaRC, and evaluate it on Alibaba Cloud via large-scale simulations and real-cloud experiments.
Authors' Contact Information: Xiaolu Li; Han Yuan; Xuan Liu; Junlong Zhang, Huazhong University of Science and Technology, Wuhan, China; Patrick P. C. Lee, The Chinese University of Hong Kong, Hong Kong, China; Yuchong Hu; Dan Feng, Huazhong University of Science and Technology, Wuhan, China.

ACM 1553-3093/2024/1-ART1
https://doi.org/0000001.0000001

1 INTRODUCTION

Erasure coding has been widely deployed in practical distributed storage systems to provide fault tolerance against data loss in failed storage nodes, while incurring significantly lower redundancy overhead than traditional replication [45]. Among many erasure codes, Reed-Solomon (RS) codes are the most popular and are reportedly deployed in production, such as Google [11], Facebook [27], Backblaze [5], and CERN [30]. However, RS codes are known to incur high *repair bandwidth* (i.e., the amount of traffic being transferred during a repair operation) when repairing a failed node, as the repair of any lost block needs to retrieve multiple coded blocks from other available nodes for decoding, leading to bandwidth amplification.

Many repair-friendly erasure codes have been proposed in the literature to reduce the repair bandwidth of RS codes. Examples include regenerating codes [10, 28, 32, 38, 43], locally repairable codes [15, 19, 37], and piggybacking codes [34, 35]. In particular, minimum-storage regenerating (MSR) codes [10] are theoretically proven to be repair-optimal, such that they minimize the repair bandwidth for repairing a single node failure while preserving the minimum storage redundancy as in RS codes (i.e., the redundancy is the minimum among any erasure code that tolerates the same number of node failures). For example, compared with the (14,10) RS code adopted by Facebook [27, 34] (i.e., 10 original uncoded blocks are encoded into 14 RS-coded blocks), MSR codes with the same coding parameters can reduce the repair bandwidth by 67.5%. Given the theoretical guarantees of MSR codes, many follow-up efforts have proposed practical constructions for MSR codes and evaluated their performance in real-world distributed storage systems (e.g., [13, 28, 32, 43]). For example, Clay codes [43] are shown to minimize both repair bandwidth and I/Os (i.e., the amount of disk I/Os to local storage during a repair operation is the same as the minimum repair bandwidth), support general coding parameters, and be deployed and integrated in Ceph [6].

While MSR codes provably minimize the repair bandwidth, their practical repair performance remains bottlenecked by the node where the lost block is decoded, as the node needs to retrieve more data from other available nodes than the amount of lost data; in other words, bandwidth amplification still exists, albeit less severely than with RS codes. To mitigate the repair bottleneck issue, recent studies [24, 26] have shown how to parallelize and load-balance the repair for RS codes across multiple available nodes by decomposing the repair operation into partial repair sub-operations that are executed in different nodes in parallel and combining the partially repaired blocks into the final decoded block. Thus, it is natural to ask whether we can also decompose and parallelize the repair for MSR codes like RS codes. Unfortunately, the answer is negative: the repair for RS codes (which belong to *scalar* codes) satisfies the additive associativity of linear combinations and the repair operation can be decomposed; in contrast, MSR codes (which belong to *vector* codes) have a different mathematical structure from RS codes, such that the repair of MSR codes needs to solve a system of linear combinations and cannot be directly decomposed (see §2 for details).

This motivates an alternative to parallelize the repair of MSR codes. Our insight is that MSR codes build on *sub-packetization*, in which a block is partitioned into sub-blocks and the repair of a lost block in MSR codes involves retrieving a subset of sub-blocks from other available nodes for decoding. The sub-blocks of a lost block can be represented as different linear combinations, and are finally decoded by solving the system of linear combinations. Based on sub-packetization, our idea is to distribute the repair of sub-blocks across different available nodes and later combine the repaired sub-blocks to reconstruct the lost block. An open question is how to schedule the parallel repair of MSR codes in order to balance the *repair load* (i.e., the amount of traffic sent or received by a node) across the available nodes.

In this paper, we present HyperParaRC, a novel parallel repair framework for MSR codes designed to balance the repair load across available nodes and hence accelerate the repair operation, while

maintaining low repair bandwidth. HyperParaRC enhances ParaRC, proposed in our conference version [22] for the parallel repair of MSR codes, in two key aspects. First, HyperParaRC significantly reduces the search time of ParaRC to find efficient repair solutions, which even outperform those returned by ParaRC in various settings. Second, while ParaRC focuses on improving single-block repair performance via intra-stripe parallelism, HyperParaRC combines both intra-stripe and inter-stripe parallelism to achieve high full-node recovery performance. To summarize, this paper makes the following contributions:

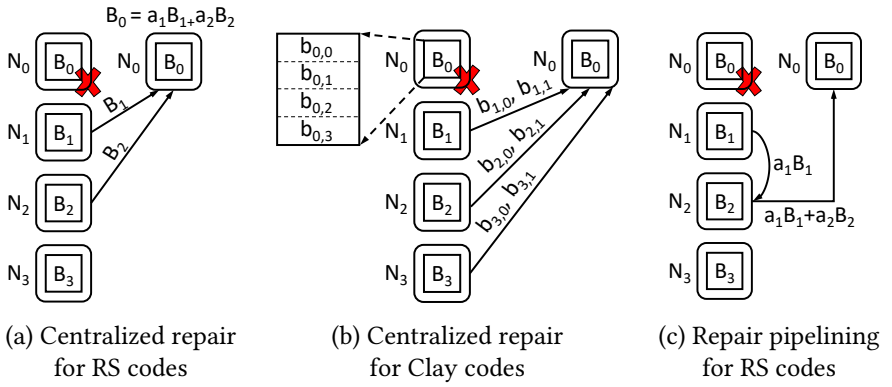
- We observe that there exists a trade-off between repair bandwidth and maximum repair load. To formally analyze the trade-off, we model the repair operation of MSR codes as a directed acyclic graph (DAG) [23] and solve the parallel repair problem as a DAG coloring problem. We identify an extreme point, the *min-max repair load (MLP) point*, which minimizes the maximum repair load with the smallest possible repair bandwidth.
- We show that finding the MLP is generally computationally expensive. Even though our earlier proposed ParaRC uses a pruning-based heuristic to efficiently identify an approximate MLP, its search time remains significant for large coding parameters (e.g., on the order of days). In this paper, we find that the repair solutions are closely related to a property called *affinity*, which captures the likelihood that a sub-block can be computed from some locally stored input sub-blocks (i.e., without incurring the bandwidth for retrieving them from other nodes), and the Pareto-optimal solutions tend to have high affinity with low bandwidth in sub-block computations. For HyperParaRC, we propose an affinity-based heuristic that quickly identifies an approximate MLP in sub-seconds for large coding parameters, significantly faster than the pruning-based heuristic in ParaRC.
- We design a full-node recovery scheduling algorithm that performs both intra-stripe and inter-stripe parallel repair scheduling based on affinity to achieve high recovery performance.
- We prototype HyperParaRC atop Hadoop 3.3.4 HDFS [3] and evaluate HyperParaRC on Alibaba Cloud [1], including large-scale simulations and real-cloud experiments. HyperParaRC reduces the single-block repair time by 68.2% compared with the centralized repair for Clay codes and by 21.3% compared with ParaRC. Also, HyperParaRC reduces the full-node recovery time of ParaRC by 45.5% via both intra-stripe and inter-stripe parallel repair scheduling. HyperParaRC reduces the full-node recovery time of the default repair method in Hadoop 3.3.4 HDFS by 70.9%. We open-source our HyperParaRC prototype at: **https://github.com/ukulililixl/hyperpararc**.

2 BACKGROUND AND MOTIVATION

2.1 Basics of Erasure Coding

We review the basics of erasure coding. We consider a large-scale distributed storage system that organizes data and performs reads/writes in fixed-size *blocks*, such that the block size is large enough (e.g., 128 MiB in Hadoop 3.3.4 HDFS [3] and 256 MiB in Facebook [33]) to mitigate I/O overhead. In this work, we target the distributed storage environments where the network bandwidth and disk I/Os are the bottlenecks, as opposed to the computational overhead for encoding and decoding operations in erasure coding.

There are many approaches to construct erasure codes, among which Reed-Solomon (RS) codes [36] are the most widely deployed (e.g., [5, 11, 27, 30]). Specifically, an (n, k) RS code, configured by two parameters k and n (where $n > k$), encodes every set of k original uncoded blocks into n coded blocks, such that any k out of n coded blocks suffice to decode the k original uncoded blocks. Each set of n coded blocks is called a *stripe*. In this work, we focus on a single stripe, while multiple stripes are independently and identically encoded. Each stripe is stored in n distinct nodes, so as to tolerate any $n - k$ node (or block) failures. RS codes satisfy three practical properties: (i) *generality*,



(a) Centralized repair for RS codes (b) Centralized repair for Clay codes (c) Repair pipelining for RS codes

Fig. 1. Repair examples: (a) conventional repair for the $(4, 2)$ RS code; (b) centralized repair for the $(4, 2)$ Clay code (which minimizes the repair bandwidth); and (c) repair pipelining for the $(4, 2)$ RS code (which minimizes the maximum repair load).

where n and k can be general parameters (provided that $n > k$), (ii) *maximum distance separable (MDS)*, where the redundancy overhead n/k is the minimum for tolerating any $n - k$ node failures, and (iii) *systematic*, where the k uncoded blocks are kept in the n coded blocks (i.e., the uncoded blocks remain directly accessible after encoding).

We elaborate on the mathematical properties of RS codes to help motivate our work. In this paper, we treat the uncoded and coded blocks equivalently in a systematic stripe and simply refer to them as ''blocks'' in our discussion. Let $B_0, B_1, \cdots, B_{n-1}$ be the n blocks of a stripe in an (n, k) RS code that are respectively stored in n nodes, denoted by $N_0, N_1, \cdots, N_{n-1}$. Each block can be expressed as a linear combination of k blocks of the same stripe under Galois Field arithmetic. For example, we have $B_0 = \sum_{i=1}^{k} a_i B_i$ for some coding coefficients a_i's $(1 \leq i \leq k)$.

Despite the popularity, RS codes are known to incur high repair penalty, since repairing a single lost block in RS codes needs to transfer multiple blocks of the same stripe from other available nodes. The repair penalty manifests in two aspects. First, the repair incurs high *repair bandwidth*, defined as the amount of traffic transferred over the network during a single-block repair operation. In general, an (n, k) RS code incurs a repair bandwidth of k times the block size when repairing a lost block. Figure 1(a) shows an example of the conventional centralized repair for the $(4, 2)$ RS code. To repair a lost block (say B_0), the new node (say N_0) downloads *any* $k = 2$ blocks (say B_1 and B_2 from N_1 and N_2, respectively), so as to repair B_0 via the linear combination of the downloaded blocks. Thus, the repair bandwidth is 2 blocks.

Second, the conventional centralized repair also incurs high *maximum repair load*, where the repair load of a node is defined as the amount of traffic that the node sends or receives, whichever is larger, during a repair operation, and the maximum repair load is the largest repair load among all nodes. In the centralized repair, the new node has the most traffic among all nodes, as it receives an amount of traffic that is k times the block size, while each other available node sends one block only. Thus, the performance of the centralized repair is bottlenecked by the new node. For example, from Figure 1(a), the new node N_0 has the most received traffic, and the maximum repair load is also 2 blocks.

Thus, the repair performance in RS codes is dominated by both repair bandwidth and maximum repair load. We argue that while many studies focus on reducing the repair bandwidth (§2.2) or reducing the maximum repair load (§2.3), there exists a trade-off in minimizing both of the performance metrics (§2.4).

In this work, we mainly consider two types of repair operations: (i) *single-block repair*, where a storage system repairs a lost block or a client issues a degraded read to a lost block, and (ii) *full-node recovery*, where a storage system repairs all lost blocks of a failed node. Both types of repair operations assume that there is only one lost block in a stripe, as single failures are the most common failure scenario in current erasure-coding deployment [15, 33]. Multiple lost blocks in a stripe are less common, but become a valid problem in wide-stripe deployment [14, 17]. We pose the analysis of multiple lost blocks for wide stripes as future work.

2.2 Reducing Repair Bandwidth

Existing studies on erasure coding reduce the repair bandwidth by proposing new erasure code constructions. Examples include regenerating codes [10, 13, 28, 32, 38, 42, 43], locally repairable codes [15, 37], and piggybacking codes [34, 35]. In this paper, we focus on *minimum-storage regenerating (MSR)* codes (first proposed in [10]), which theoretically minimize the repair bandwidth for repairing a single lost block, with the minimum redundancy (i.e., MDS property) as RS codes.

MSR codes differ from RS codes by performing *sub-packetization*, which divides a block into multiple sub-blocks and performs encoding and repair at the sub-block granularity. Specifically, an (n, k) MSR code partitions each block B_i ($0 \leq i \leq n-1$) into w sub-blocks ($w > 1$), denoted by $b_{i,0}, b_{i,1}, \cdots, b_{i,w-1}$, such that each sub-block is encoded through a linear combination of $k \times w$ sub-blocks from k blocks (under Galois Field arithmetic). To repair any lost block (or w sub-blocks therein), MSR codes only transfer sub-blocks from the other nodes, such that the total amount of traffic of the transferred sub-blocks is minimized.

Classical MSR codes [10] require that the available nodes read all their locally stored sub-blocks, encode them, and transfer the encoded sub-blocks to the new node (with the minimum repair bandwidth) to repair the lost block. In this work, we consider two state-of-the-art MSR codes, namely Clay codes [43] and Butterfly codes [28], both of which have been implemented and empirically evaluated. Our goal is to show the applicability of our work to different MSR codes, using Clay codes and Butterfly codes as two representatives. In particular, Clay codes minimize both repair bandwidth and I/Os (a.k.a. repair-by-transfer [38]) for general coding parameters n and k, while Butterfly codes minimize both repair bandwidth and I/Os for the k uncoded blocks in a systematic stripe and support $n - k = 2$ only. Thus, we use Clay codes as our major baseline throughout the paper.

We first provide an overview of Clay codes. At a high level, Clay codes repair a lost block in three steps: (i) *pairwise reverse transformation (PRT)*, which couples sub-blocks in pairs and generates intermediate sub-blocks; (ii) *MDS decoding*, which performs linear combinations on k sub-blocks to decode intermediate sub-blocks and a subset of repaired sub-blocks; and (iii) *pairwise forward transformation (PFT)*, which again couples sub-blocks in pairs to generate the remaining repaired sub-blocks, such that the lost block is completely repaired. In Clay codes, the number of sub-blocks w is given by $w = (n-k)^{\lceil n/(n-k) \rceil}$.

Let us take the $(4, 2)$ Clay code (where $w = 4$) as an example, as shown in Figure 1(b). Let c_i be the i^{th} intermediate sub-block generated in the repair. Also, let $\langle \ldots \rangle_i$ denote some linear combination of sub-blocks within the brackets, where the subscript i differentiates the linear combinations with different coding coefficients. To repair a lost block, say B_0, the new node N_0 downloads two sub-blocks $b_{i,0}$ and $b_{i,1}$ from each N_i, where $1 \leq i \leq 3$. N_0 repairs the four sub-blocks of B_0 as follows. First, in the PRT step, N_0 generates two intermediate sub-blocks c_0 and c_1 by coupling $b_{2,1}$ and $b_{3,0}$:

$$c_0 = \langle b_{2,1}, b_{3,0} \rangle_0, \quad c_1 = \langle b_{2,1}, b_{3,0} \rangle_1. \tag{1}$$

Second, in the MDS decoding step, N_0 performs linear combinations on $b_{2,0}$ and c_0, and on $b_{3,1}$ and c_1. It repairs $b_{0,0}$ and $b_{0,1}$, and generates two intermediate sub-blocks c_2 and c_3:

$$\begin{aligned} b_{0,0} &= \langle b_{2,0}, c_0 \rangle_2, \quad & c_2 &= \langle b_{2,0}, c_0 \rangle_3, \\ b_{0,1} &= \langle b_{3,1}, c_1 \rangle_4, \quad & c_3 &= \langle b_{3,1}, c_1 \rangle_5. \end{aligned} \tag{2}$$

Finally, in the PFT step, N_0 repairs $b_{0,2}$ by coupling $b_{1,0}$ and c_2, and repairs $b_{0,3}$ by coupling $b_{1,1}$ and c_3:

$$b_{0,2} = \langle b_{1,0}, c_2 \rangle_6, \quad b_{0,3} = \langle b_{1,1}, c_3 \rangle_7. \tag{3}$$

The $(4, 2)$ Clay code minimizes the repair bandwidth to 1.5 blocks (it downloads six sub-blocks). Compared with the $(4, 2)$ RS code, the $(4, 2)$ Clay code reduces the repair bandwidth by 25%. Note that the maximum repair load of the Clay code is also 1.5 blocks (same as the repair bandwidth), which is the amount of traffic downloaded in the new node.

We also consider Butterfly codes [28] in this paper. For an (n, k) Butterfly code ($n - k = 2$), we focus on the repair of the first k original uncoded blocks in a systematic stripe (whose repair bandwidth and I/Os are both minimized). An (n, k) Butterfly code divides each block into $w = 2^{k-1}$ sub-blocks. When repairing a lost block, a new node first downloads half of the sub-blocks from each available node. It then selects different subsets of sub-blocks among all the received sub-blocks and performs XOR operations to repair the w sub-blocks of the lost block. For example, to repair a lost block for the $(4, 2)$ Butterfly code, the new node downloads half of the sub-blocks from each of the three available nodes, such that both repair bandwidth and maximum repair load are 1.5 blocks.

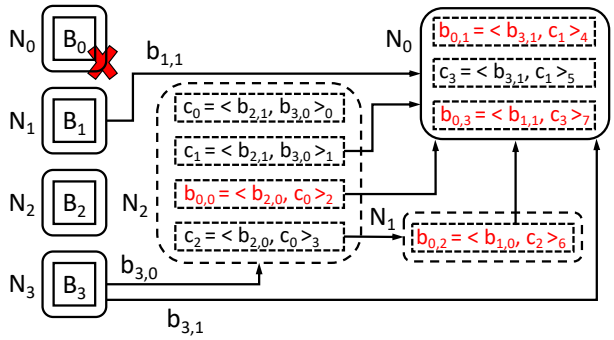
2.3 Reducing Maximum Repair Load

Some studies reduce the maximum repair load by decomposing and parallelizing a repair operation across the available nodes [24, 26]. In this work, we focus on *repair pipelining* [24], which reduces the time of repairing a lost block to almost the same as the time of directly reading a block.

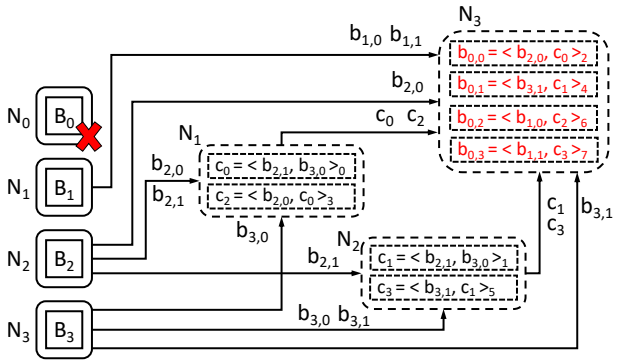
Repair pipelining is mainly designed for RS codes [36]. It divides a single-block repair operation into multiple sub-block repair operations and evenly distributes sub-block repair operations across all nodes. For example, suppose that we use repair pipelining to repair a lost block B_0 for an (n, k) RS code. It first divides each block B_i $(0 \le i \le n-1)$ into multiple sub-blocks, denoted by $b_{i,0}, b_{i,1}, \cdots$. Recall that each block can be expressed as a linear combination of k blocks (§2.1), say $B_0 = \sum_{i=1}^{k} a_i B_i$ for some coding coefficients a_i's. Repair pipelining makes two observations. First, each sub-block in B_0 is also a linear combination of the k sub-blocks at the same block offset with the same coding coefficients, i.e., $b_{0,j} = \sum_{i=1}^{k} a_i b_{i,j}$, for the j-th sub-block. Second, the linear combination is addition associative, meaning that $b_{0,j}$ can be computed from the linear combinations of partial terms.

To repair B_0, repair pipelining works as follows. First, N_1 starts the repair of $b_{0,0}$ by sending $a_1 b_{1,0}$ from its local storage to N_2. Second, N_2 combines the received $a_1 b_{1,0}$ with $a_2 b_{2,0}$ from its local storage to form $a_1 b_{1,0} + a_2 b_{2,0}$. Third, N_2 sends $a_1 b_{1,0} + a_2 b_{2,0}$ to N_3; meanwhile, N_1 can start the repair of $b_{0,1}$ by sending $a_1 b_{1,1}$ from its local storage to N_2 without interfering with N_2's transmission. Finally, the last available node N_k reconstructs $b_{0,j}$ for each j-th sub-block and sends $b_{0,j}$ to N_0.

Repair pipelining reduces the maximum repair load to the same as the block size. For example, Figure 1(c) shows an example of repair pipelining for the $(4, 2)$ RS code. The maximum repair load is only 1 block since each of the k available nodes sends or receives one block of data; it is even less than that in Clay codes (Figure 1(b)). Note that the repair bandwidth remains 2 blocks, the same as in the conventional repair for RS codes (Figure 1(a)), since k available nodes transfer k blocks of data in total.



(a) Repair bandwidth = 1.75 blocks; Max. repair load = 1.25 blocks



(b) Repair bandwidth = 3.25 blocks; Max. repair load = 2 blocks

Fig. 2. Examples of the parallel repair for the $(4, 2)$ Clay code. The example in figure (a), with more careful repair scheduling, has both less repair bandwidth and less maximum repair load than the example in figure (b).

2.4 Motivation and Challenges

From §2.3, a natural question to ask is whether we can apply repair pipelining to MSR codes (§2.2) to reduce the maximum repair load. Unfortunately, the answer is negative, mainly because the repair of sub-blocks is not based on the addition associativity as in RS codes; instead, it is done by solving a system of linear combinations (e.g., see Equations (1)-(3) in §2.2 for Clay codes). Thus, we cannot pipeline the repair of individual sub-blocks of MSR codes as in RS codes.

Nevertheless, the sub-packetization nature of MSR codes offers an opportunity for parallelizing a repair operation to reduce the maximum repair load. First, the repair of a sub-block in MSR codes only requires a subset of available sub-blocks; for example, in the $(4, 2)$ Clay code, each sub-block is a linear combination of two currently stored or intermediate sub-blocks. Thus, we can distribute the repair operations of sub-blocks across different nodes for load balancing. Second, in erasure coding implementation, each block is further divided into smaller-sized units (called *packets*), so that the repair of a block can be parallelized at the packet level (see §6 for implementation details).

Figure 2(a) shows a parallel repair example for the $(4, 2)$ Clay code. First, in the PRT step, N_2 generates c_0 and c_1 from $b_{3,0}$ (retrieved from N_3) and $b_{2,1}$ (locally stored in N_2). Second, in the MDS decoding step, N_2 decodes c_2 and $b_{0,0}$ from $b_{2,0}$ (locally stored in N_2) and c_0 (generated in the PRT step), while N_0 generates c_3 and $b_{0,1}$ from c_1 (retrieved from N_2) and $b_{3,1}$ (retrieved from N_3). Finally, in the PFT step, N_1 repairs $b_{0,2}$ from $b_{1,0}$ (locally stored in N_1) and c_2 (retrieved from N_2), while N_0 repairs $b_{0,3}$ from $b_{1,1}$ (retrieved from N_1) and c_3 (generated in the MDS decoding step).

	Repair bandwidth	Maximum repair load
RS; centralized	2 blocks (highest)	2 blocks (highest)
Clay; centralized	1.5 blocks (lowest)	1.5 blocks (high)
RS; parallel	2 blocks (highest)	1 block (lowest)
Clay; parallel	1.75 blocks (medium)	1.25 blocks (medium)

Table 1. Summary of the four repair methods for $(n, k) = (4, 2)$.

Also, N_0 retrieves the repaired $b_{0,0}$ and $b_{0,2}$ from N_2 and N_1, respectively. In this example, the repair operation can be parallelized in two aspects: (i) the repair of $b_{0,1}$ and $b_{0,3}$ in N_0, as well as the repair of $b_{0,2}$ in N_1, can be performed in parallel; and (ii) the sub-block repair operations in N_0, N_1, and N_2 can be parallelized at the packet level. Thus, the maximum repair load is 1.25 blocks (i.e., the five sub-blocks $b_{0,0}$, $b_{0,2}$, $b_{1,1}$, $b_{3,1}$, and c_1 retrieved by N_0).

Such a parallel repair approach may amplify the repair bandwidth, as some sub-blocks are reused more than once by different nodes. For example, the sub-blocks $b_{2,1}$ and $b_{3,0}$ are used to compute c_1, c_2, and $b_{0,0}$. Each of the three sub-blocks will be transmitted over the network. Thus, instead of transmitting each of the sub-blocks $b_{2,1}$ and $b_{3,0}$ only once as in the centralized repair (Figure 1(b)), the parallel repair now includes the sub-blocks $b_{2,1}$ and $b_{3,0}$ in three transmissions. The repair bandwidth increases from the minimum point of 1.5 blocks to 1.75 blocks.

How to carefully schedule the parallel repair of different sub-blocks is a critical issue. Figure 2(b) shows another example of the parallel repair of the $(4, 2)$ Clay code, where the repair is less efficiently scheduled. In this example, the sub-blocks $b_{2,0}$, $b_{2,1}$, and $b_{3,0}$ are all transmitted twice. Thus, the repair bandwidth is 3.25 blocks, while the maximum repair load is 2 blocks.

In summary, the parallel repair of MSR codes can be scheduled to balance the trade-off between repair bandwidth and maximum repair load, as shown in Table 1 for the $(4, 2)$ Clay code. Our goal in this paper is to design a parallel repair framework that can effectively balance the trade-off for general coding parameters of MSR codes.

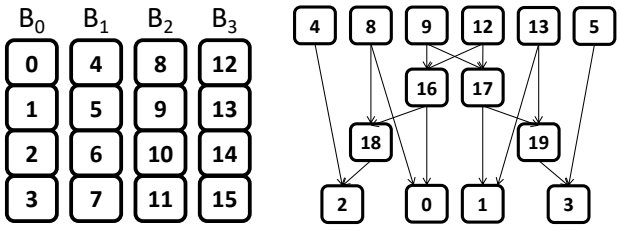
3 MODEL AND ANALYSIS

Before we design the parallel repair framework for MSR codes, we first formulate a generic repair model that characterizes the trade-offs between repair bandwidth and maximum repair load for different repair solutions, either centralized (e.g., Figures 1(a) and 1(b)) or parallel (e.g., Figures 1(c) and 2). In this section, we design our repair model (§3.1) and evaluate both repair bandwidth and maximum repair load for a repair solution (§3.2). Finally, we analyze the trade-offs between repair bandwidth and maximum repair load for different repair solutions on RS and MSR codes (§3.3).

3.1 Characterizing Repair Solutions

Design requirements. We first identify three design requirements for our repair model to characterize repair solutions based on our example in Figure 2:

- R1: It can describe the linear combination relationships of sub-blocks (e.g., $b_{0,0}$ is the linear combination of $b_{2,0}$, $b_{2,1}$, and $b_{3,0}$).
- R2: It can describe which node is scheduled to execute a repair operation for each sub-block and how the repair operation is executed (e.g., N_2 downloads $b_{3,0}$ from N_3 and generates $b_{0,0}$ with its locally stored $b_{2,0}$ and $b_{2,1}$).
- R3: It can describe how the repaired sub-blocks are collected (e.g., $b_{0,0}$, $b_{0,1}$, $b_{0,2}$, and $b_{0,3}$ can be repaired in different nodes, but are finally collected by N_0 for reconstructing block B_0).



(a) Layout (b) ECDAG

Fig. 3. An ECDAG example of repairing B_0 using the $(4, 2)$ Clay code ($w = 4$).

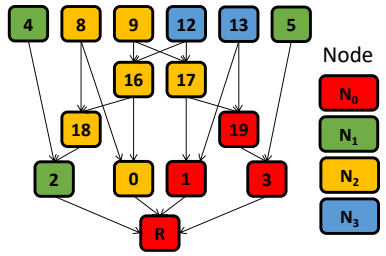
Our repair model builds on the ECDAG abstraction [23], which characterizes and schedules erasure coding operations in distributed storage systems. Note that an ECDAG can model the linear combination relationships of sub-blocks (i.e., R1 addressed), but cannot directly schedule the repair operations for different sub-blocks in different nodes (i.e., R2 and R3 not addressed). In the following, we first introduce the ECDAG abstraction, and then explain how it can be extended to address all our requirements.

Basics of an ECDAG. We provide an overview of an ECDAG. An ECDAG $G = (V, E)$ is a directed acyclic graph (DAG) that describes an erasure coding operation (including the repair of a block), where V is the set of vertices and E is the set of edges. A vertex $v_\ell \in V$ (where $\ell \geq 0$) refers to either a sub-block that is stored in a node (i.e., $\ell = i \times w + j$ for $b_{i,j}$, where $i, j \geq 0$) or an intermediate sub-block that is generated on-the-fly but will not be finally stored (i.e., $\ell \geq n \times w$). With a slight abuse of notation, we refer to a sub-block with its vertex v_ℓ, where ℓ is the index. An edge $e(\ell_1, \ell_2) \in E$ means that the sub-block v_{ℓ_1} is an input to the linear combination for computing the sub-block v_{ℓ_2}. We refer to v_{ℓ_1} as an *input vertex* of v_{ℓ_2}, and v_{ℓ_2} as an *output vertex* of v_{ℓ_1}. Note that the repair workflows vary across blocks, so the repair of each block will lead to a different ECDAG instance.

We use Clay codes [43] as an example to show how an ECDAG describes its repair workflow. Figure 3(a) shows the block layout of the $(4, 2)$ Clay code (where $w = 4$) in an ECDAG, and Figure 3(b) shows the repair flow for block B_0, which we introduce in §2.2. First, in the PRT step, we couple sub-blocks v_9 ($b_{2,1}$) and v_{12} ($b_{3,0}$) as a pair and perform linear combinations to generate two intermediate sub-blocks v_{16} (c_0) and v_{17} (c_1). Second, in the MDS decoding step, we decode sub-blocks v_0 ($b_{0,0}$) and v_{18} (c_2) from sub-blocks v_8 ($b_{2,0}$) and v_{16} (c_0), and we decode sub-blocks v_1 ($b_{0,1}$) and v_{19} (c_3) from sub-blocks v_{13} ($b_{3,1}$) and v_{17} (c_1). Note that the sub-blocks v_0 ($b_{0,0}$) and v_1 ($b_{0,1}$) of B_0 are repaired. Finally, in the PFT step, we couple sub-blocks v_4 ($b_{1,0}$) and v_{18} (c_2) to repair sub-block v_2 ($b_{0,2}$), and also couple sub-blocks v_5 ($b_{1,1}$) and v_{19} (c_3) to repair sub-block v_3 ($b_{0,3}$). B_0 is now fully repaired.

pECDAG. We extend the ECDAG abstraction into the pECDAG abstraction to support the scheduling of parallel sub-block repair operations, so that we can model the trade-off between repair bandwidth and maximum repair load. Specifically, a pECDAG makes two extensions over an ECDAG. First, it associates each vertex with a *color* that corresponds to a node, such that the node is responsible for generating or storing all sub-blocks associated with the same-colored vertices (i.e., R2 addressed). Second, it connects all repaired sub-blocks, which may reside in different nodes, to a vertex R, which represents a data collector (i.e., R3 addressed). Figure 4(a) shows the pECDAG for the parallel repair in Figure 2.

For example, from Figure 4(a), N_2 (i.e., yellow-colored) computes the sub-block v_{17} (c_1) in Figure 2 and sends it to N_0 (i.e., red-colored), which repairs the sub-blocks v_1 ($b_{0,1}$) and v_3 (i.e., $b_{0,3}$). Also, N_2



(a) pECDAG

	In	Out
N_0	5	0
N_1	1	2
N_2	1	3
N_3	0	2

(b) Traffic table

Fig. 4. A pECDAG example of $(4, 2)$ Clay code with $w = 4$ to repair B_0.

computes the sub-blocks v_0 ($b_{0,0}$) and v_{18} (c_2). It sends v_{18} to N_1 (i.e., green-colored), which repairs the sub-block v_2 ($b_{0,2}$). Finally, N_0 collects all the repaired sub-blocks for the reconstruction of B_0.

To help our discussion, we refer to the topmost vertices (i.e., v_4, v_5, v_8, v_9, v_{12}, and v_{13} in Figure 4(a)) as the *leaf vertices*, which correspond to sub-blocks in available nodes. We refer to the vertex R as the *root vertex*. We refer to the remaining vertices (i.e., v_0, v_1, v_2, v_3, v_{16}, v_{17}, v_{18}, v_{19} in Figure 4(a)) as *intermediate vertices*, which represent intermediate sub-blocks generated in a repair operation. Note that the colors of leaf vertices are determined by the nodes that store available blocks, while the color of the root vertex is determined by the node selected to persist the repaired block. Different color combinations of the intermediate vertices can lead to different repair solutions for repairing a lost block.

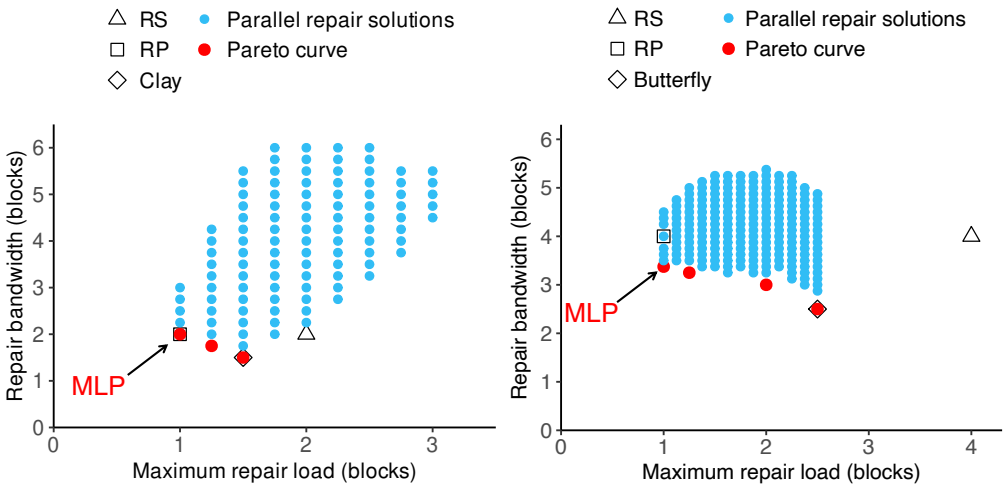
3.2 Evaluating Repair Solutions

Given (n, k, w) and the block to repair, there are different ways to color the vertices of a pECDAG, so there are multiple possible pECDAG instances that correspond to different repair solutions for repairing a single lost block. We associate each pECDAG instance with a *traffic table*, so as to efficiently quantify the repair bandwidth and maximum repair load of the corresponding repair solution.

Definition of a traffic table. A traffic table maintains the amount of data that each node sends or receives when repairing a block. For each node in the system, the traffic table records the number of incoming sub-blocks received by the node and the number of outgoing sub-blocks sent by the node. The repair bandwidth is the total number of incoming sub-blocks (or equivalently, the total number of outgoing sub-blocks) of all nodes, while the maximum repair load is the largest number of incoming or outgoing sub-blocks of a node across all nodes. For example, Figure 4(b) shows the traffic table for the parallel repair solution shown in Figure 2, in which the repair bandwidth is 7 sub-blocks and the maximum repair load is 5 sub-blocks.

Construction of a traffic table. We show how we generate the traffic table for a given pECDAG instance. We initialize a traffic table T with two arrays $T.In$ and $T.Out$, which record the numbers of incoming and outgoing sub-blocks for each node, respectively. For each vertex v_i, we traverse each edge $e(v_i, v_j)$. Let N' and N'' be two nodes with respect to the colors of v_i and v_j, respectively. If v_i and v_j have different colors, we increment $T.Out[N']$ and $T.In[N'']$ by one; however, if there exist two edges, say $e(v_i, v_j)$ and $e(v_i, v_h)$, such that v_j and v_h have the same color that is different from v_i's color, we only increment T once for the corresponding pairs of nodes. The rationale is that the sub-block v_i only needs to be transmitted once to calculate the sub-blocks v_j and v_h.

For example, in Figure 4(a), both v_{18} and v_0 have the same color as v_8, we do not need to update the traffic table. For v_{17}, as v_1 has a different color, we count $e(v_{17}, v_1)$ as a transmission and increment



(a) $(4, 2)$ Clay code ($w = 4$) (b) $(6, 4)$ Butterfly code ($w = 8$)

Fig. 5. Trade-off analysis between repair bandwidth and maximum repair load.

the traffic table. As v_{19} and v_1 have the same color, we do not need to increment the traffic table for $e(v_{17}, v_{19})$.

3.3 Trade-off Analysis

Based on a pECDAG and its traffic table, we study the trade-off between repair bandwidth and maximum repair load for repair solutions. Our idea is to enumerate all possible color combinations of a pECDAG (i.e., all possible repair solutions for repairing a single lost block) and find the corresponding traffic table for each color combination. Note that the colors of the leaf vertices and the root vertex are fixed (§3.1). Thus, for a pECDAG, we only need to enumerate the color combinations for the intermediate sub-blocks and repaired sub-blocks. Currently, we assume that the repair operation of a stripe is scheduled among the nodes (i.e., n nodes for an (n, k) code) that store the blocks of the stripe, so as to limit the interference across different stripes.

We consider the repair scenarios of two MSR codes: the $(4, 2)$ Clay code and the $(6, 4)$ Butterfly code. We consider the repair of block B_0 (i.e., the first block of a stripe) and construct a pECDAG for each of them. We apply a brute-force search to enumerate all color combinations; for each color combination, we generate the traffic table and obtain the corresponding repair bandwidth and maximum repair load. We show the spectrum of repair bandwidth and maximum repair load for different color combinations under the $(4, 2)$ Clay code (Figure 5(a)) and the $(6, 4)$ Clay code (Figure 5(b)). In the figures, we highlight the points corresponding to the centralized repair for RS codes (RS), repair pipelining for RS codes (RP), and the centralized repair for Clay codes (Clay) or Butterfly codes (Butterfly) for comparisons.

We find that different color combinations for an MSR code present different trade-offs between repair bandwidth and maximum repair load. Among all the color combinations, we focus on the *Pareto-optimal* points (i.e., the red points in Figure 5), which represent the cases that cannot further reduce the repair bandwidth (resp. maximum repair load) without increasing the maximum repair load (resp. repair bandwidth). Among the Pareto-optimal points, we define the *min-max repair load point (MLP)*, which minimizes the maximum repair load, and whose repair bandwidth is minimized given this optimal maximum repair load. Note that the MLP does not guarantee the absolute minimum repair bandwidth. For example, for the $(4, 2)$ Clay code, the MLP coincides with the point

of RP; for the $(6, 4)$ Butterfly code, the MLP reduces the repair bandwidth by 15.6% compared with RP, while achieving the same maximum repair load as RP.

This observation indicates that the parallel repair of an MSR code may further improve the repair performance of a distributed storage system if we can find the MLP. However, it is non-trivial to find the MLP in general. While the brute-force approach can always find the MLP, it also has high complexity. For a pECDAG of an (n, k) MSR code with w sub-blocks in a block, the lower bound of the number of vertices being colored is w (i.e., when there is no intermediate sub-block, we only need to color the w repaired sub-blocks). In this case, the lower bound of the total number of color combinations is n^w. For Clay codes, the lower bound is $n^{(n-k)^{\lceil n/(n-k) \rceil}}$, while for Butterfly codes, the lower bound is $n^{2^{k-1}}$. For example, for the $(14, 10)$ Clay code, the number of color combinations is no less than 14^{256}, while for the $(12, 10)$ Butterfly code, the number of combinations is no less than 12^{512}, which are not solvable in polynomial time. Thus, for reasonably large (n, k), it is important to reduce the running time of finding the MLP.

4 HEURISTIC DESIGN

As the brute-force approach is generally time-consuming for finding the MLP, especially for large coding parameters, we propose to design a heuristic to find an approximate MLP that is close to the MLP. Our goal is to find an efficient parallel repair solution represented in a pECDAG that keeps both repair bandwidth and maximum repair load as low as possible.

In our conference version [22], we design a pruning-based heuristic (§4.1) that significantly reduces the search space compared with the brute-force approach, yet it still incurs substantial running time. In this section, we analyze the trade-off points from §3.3 and identify a property, called *affinity*, which is correlated with repair bandwidth and maximum repair load (§4.2). Finally, we propose an affinity-based heuristic that can quickly find an approximate MLP and its corresponding repair solution (§4.3). While our discussion in §4.1-§4.3 focuses on Clay codes [43], we also show how the heuristics are applied to Butterfly codes [28] (§4.4). Finally, we summarize our main insights from the heuristics (§4.5).

4.1 Pruning-based Heuristic

We first provide an overview of the pruning-based heuristic [22]; note that the pECDAG shown in Figure 4(a) (§3.1) is generated by the pruning-based heuristic to repair the block B_0 for $(4, 2)$ Clay code. Its high-level idea is to search all the color combinations for a pECDAG, while pruning some branches to reduce the search space. Intuitively, the pruning-based heuristic can be viewed as searching for the solution based on Pareto optimality, such that it searches for the MLP on the Pareto curve and prunes the dominated solutions that have both larger repair bandwidth and larger maximum repair load than a solution in the Pareto curve.

We define an *un-searched pool*, which keeps the pECDAGs that will be searched, and a *candidate pool*, which records the candidate pECDAG solutions that may be returned. At the beginning, we generate a random pECDAG, in which the color of each intermediate vertex is randomly selected from a set of candidate colors that represent the nodes storing the available blocks and the node selected for storing the repaired block. We add the random pECDAG to the un-searched pool and the candidate pool for initialization.

We iteratively retrieve a pECDAG from the un-searched pool. Each time we retrieve a pECDAG, we enumerate all the neighbors of this pECDAG, where each neighbor is generated by changing the color of only one intermediate vertex of the pECDAG that we retrieved. We compare each neighbor with the pECDAGs in the candidate pool, and add it into the candidate pool if it has lower repair bandwidth or lower maximum repair load than the existing pECDAGs in the candidate pool. Also,

we remove an existing pECDAG from the candidate pool if it has both higher repair bandwidth and higher maximum repair load than a newly added pECDAG. For the pECDAGs that have been added to the candidate pool, we also add them to the un-searched pool for our future search.

The heuristic stops when the un-searched pool is empty. Then, we select the pECDAG with the minimum maximum repair load in the candidate pool as an approximate MLP.

While the pruning-based heuristic significantly reduces the search space compared with the brute-force approach, it does not provide any guarantee of how much running time can be reduced. It still incurs substantial running time for large coding parameters; for example, it takes 57.2 hours (over two days) for the $(14, 10)$ Clay code (§7). Also, the repair solution returned by the pruning-based heuristic applies only to the single-block repair of a single stripe. For full-node recovery involving multiple lost blocks of a failed node, we need to find a separate repair solution for each lost block, which corresponds to different stripes, and the search time can further increase.

4.2 Analysis for pECDAGs

We explore a new heuristic that identifies an approximate MLP with an algorithmic running time guarantee. Before designing the new heuristic, we first analyze the pECDAGs with different color combinations to understand the properties that enable pECDAGs to achieve both lower repair bandwidth and lower maximum repair load.

Consider two vertices v_i and v_j of a pECDAG, where v_i is an input vertex of v_j. If both vertices have the same color, then the generation of the sub-block v_j does not incur the transfer of the input sub-block v_i; otherwise, the input sub-block v_i must be transferred from one node to another, increasing both repair bandwidth and maximum repair load. Our intuition is that having the same color for both v_i and v_j is important to reduce repair bandwidth and maximum repair load. We validate our intuition and examine more color combinations by considering the colors of the intermediate vertices and the root vertex of a pECDAG. Note that we do not consider leaf vertices, which do not have any input vertex.

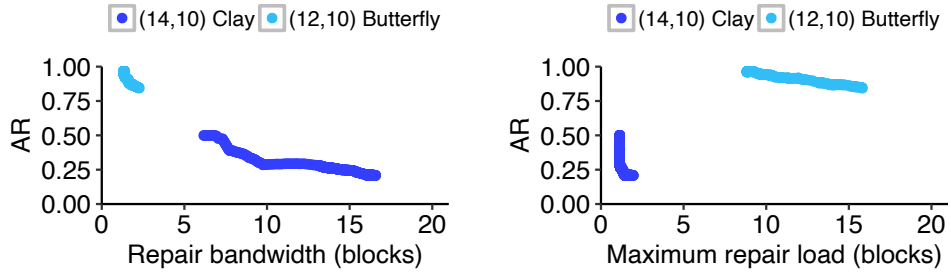
Affinity. We define a property, called *affinity*, to describe if a vertex shares the color with some of its input vertices. Specifically, for a vertex v, if its color is the same with at least one of its input vertices, we say that v has affinity. For example, in Figure 4(a), v_{16} has affinity as it shares the same color with v_9. The root vertex R also has affinity. On the other hand, v_{19} does not have affinity, as its color differs from both of its input vertices v_{13} and v_{17}.

We now define the *affinity ratio (AR)* of a pECDAG to describe the fraction of vertices that have affinity over all intermediate vertices and the root vertex. For example, in Figure 4(a), there are seven vertices with affinity, including v_{16}, v_{17}, v_{18}, v_2, v_0, v_3, and R, over all eight intermediate vertices and the root vertex. Thus, the AR of the pECDAG in Figure 4(a) is $\frac{7}{9} = 0.78$.

AR analysis. We analyze the ARs of different pECDAGs for the $(14, 10)$ Clay code and the $(12, 10)$ Butterfly code. For each code, we collect the pECDAGs retrieved from the un-searched pool during the execution of our pruning-based heuristic (§4.1) in the search for an approximate MLP. In total, we have collected 9,108 and 5,900 pECDAGs for the $(14, 10)$ Clay code and the $(12, 10)$ Butterfly code, respectively. For each pECDAG, we calculate the repair bandwidth, the maximum repair load, and the corresponding AR.

Figure 6 shows how the AR changes with respect to repair bandwidth and maximum repair load; for clarity, we only sample 10% of the collected pECDAGs to show their results instead of plotting all data points in the figures. From both Figures 6(a) and 6(b), we observe that there is a clear decreasing trend of the AR as both repair bandwidth and maximum repair load increase.

We also measure the Pearson correlation coefficients [29] between the AR and the repair bandwidth and between the AR and the maximum repair load over all collected pECDAGs, based on the



(a) AR vs. repair bandwidth (b) AR vs. maximum repair load

Fig. 6. AR analysis for the (14,10) Clay code and the (12,10) Butterfly code.

	(14, 10) Clay	(12, 10) Butterfly
cor(repair bandwidth, AR)	-0.96	-0.90
cor(maximum repair load, AR)	-0.61	-0.98

Table 2. Pearson correlation coefficient analysis for the (14, 10) Clay code and the (12, 10) Butterfly code.

following equation:

$$cor(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}}, \tag{4}$$

where $\bar{X}$ and $\bar{Y}$ denote the expectations of X and Y, respectively. Table 2 shows a negative correlation of the AR with respect to both repair bandwidth and maximum repair load for both the $(14, 10)$ Clay code and the $(12, 10)$ Butterfly code.

From the above analysis, a pECDAG with a high AR has a high likelihood of reducing both repair bandwidth and maximum repair load. Our goal is to design a heuristic to find a pECDAG with a high AR.

4.3 Affinity-based Heuristic

Based on our findings in §4.2, we propose an affinity-based heuristic to find an approximate MLP in polynomial running time. Unlike the pruning-based heuristic (§4.1), whose goal is to search for a suitable pECDAG within a pool of pECDAGs, the affinity-based heuristic aims to properly generate colors for a suitable pECDAG. Its main idea is to always select one of the colors of the input vertices for each intermediate vertex in a pECDAG, thereby increasing the AR. Specifically, the affinity-based heuristic comprises three steps.

Step 1: Initialization. We first initialize a pECDAG G, in which each leaf vertex is associated with a color based on where its corresponding available sub-block is stored, and the root vertex is also associated with a color based on where the repaired block is stored. All the intermediate vertices have unassigned colors. We also initialize an empty traffic table T with zero input and output traffic.

Step 2: Topological sorting. We perform a topological sorting on the intermediate vertices in G. For example, for the pECDAG in Figure 4(a), the sorted sequence of intermediate vertices is: v_{16}, v_{17}, v_{18}, v_0, v_1, v_{19}, v_2, and v_3.

Step 3: Affinity-based coloring. We iteratively associate a color with each intermediate vertex based on the topological ordering. Specifically, for each intermediate vertex v, we examine the set

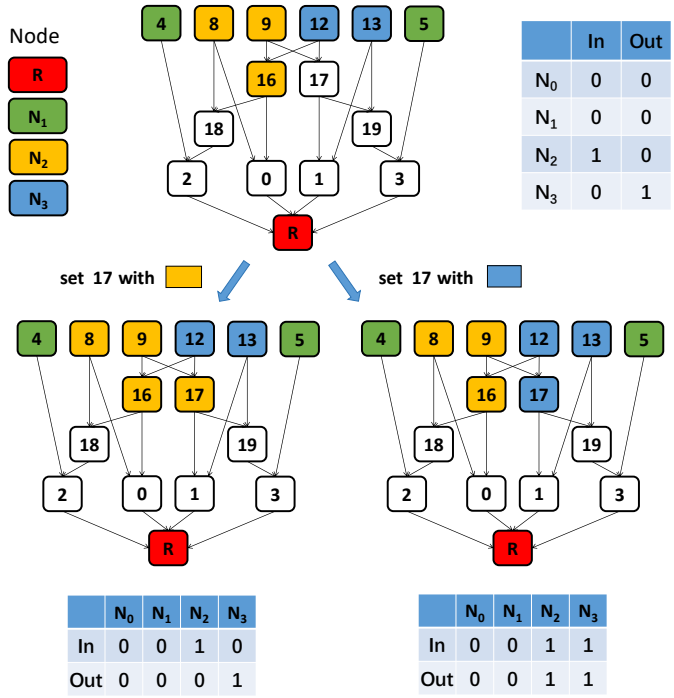


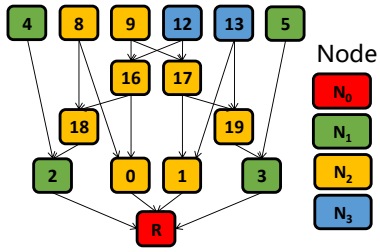
Fig. 7. Example of associating colors with intermediate vertices in the affinity-based heuristic.

(denoted by C) of all current colors of its input vertices. For each color $c \in C$, we compute the corresponding repair bandwidth and maximum repair load if c is associated with v. Among all colors in C, we select the color that minimizes the maximum repair load; if a tie occurs, we select the color that also minimizes the repair bandwidth; if a tie occurs again, we randomly select a color from the tie. Once we identify the color associated with v, we also update the traffic table T for the computations of repair bandwidth and maximum repair load in the next iteration. We repeat the process for each intermediate vertex in the topological ordering until all intermediate vertices are processed.

Example. We first show how to assign a color to a vertex in the $(4, 2)$ Clay code using the affinity-based heuristic. Then, we show how the complete pECDAG is generated by this heuristic.

Figure 7 shows an example of how the affinity-based heuristic associates intermediate vertices with colors. Suppose that we start right after v_{16} has been associated with a yellow color corresponding to N_2, meaning that N_3 will send a sub-block to N_2 (i.e., In$[N_2]$ and Out$[N_3]$ have a value one, while other entries a value zero). Based on the topological ordering, we now associate v_{17} with a color. Note that v_{17} has two input vertices, v_9 and v_{12}, in which v_9 is associated with a yellow color corresponding to N_2, and v_{12} is associated with a blue color corresponding to N_3. Thus, we can associate v_{17} with either a yellow or blue color. If we associate v_{17} with a yellow color, both repair bandwidth and maximum repair load remain unchanged. However, if we associate v_{17} with a blue color, the maximum repair load remains one, but the repair bandwidth increases to two since the sub-block for v_9 will be transferred from N_2 to N_3. Thus, we associate v_{17} with a yellow color.

Figure 8(a) shows a complete example of the pECDAG generated by the affinity-based heuristic. Compared to the pECDAG generated by the pruning-based heuristic shown in Figure 4(a) (§3.1), the colors assigned to vertices v_{19}, v_1, and v_3 are different. In Figure 4(a), the vertices are all red



(a) pECDAG

	In	Out
N_0	4	0
N_1	2	2
N_2	2	4
N_3	0	2

(b) Traffic table

Fig. 8. A pECDAG example of $(4, 2)$ Clay code with $w = 4$ to repair B_0 with affinity-based heuristic.

(i.e., R computes sub-blocks v_{19}, v_1, and v_3). In Figure 8(a), v_{19} and v_1 are yellow (i.e., N_2 computes sub-blocks v_{19} and v_1), while v_3 is green (i.e., N_1 computes the sub-block v_3). Also, Figure 8(b) shows that the affinity-based heuristic reduces the maximum repair load from 5 sub-blocks (Figure 4(b)) to 4 sub-blocks.

Complexity analysis. We now analyze the computational time complexity of the affinity-based heuristic. We consider a pECDAG with V vertices and E edges. Let α be the maximum number of input vertices of any vertex in the pECDAG (i.e., α is the maximum number of sub-blocks needed to generate a sub-block during a repair operation); note that we have $\alpha < V$ in general. In Step 1, we directly associate a color for each of the leaf vertices and the root vertex, resulting in a complexity of $O(V)$. In Step 2, we perform topological sorting, whose complexity is $O(V + E)$. In Step 3, the number of colors that we examine for each intermediate vertex is at most α, resulting in a complexity of $O(\alpha V)$. Thus, the overall complexity of the affinity-based heuristic is $O((\alpha+2)V+E)$.

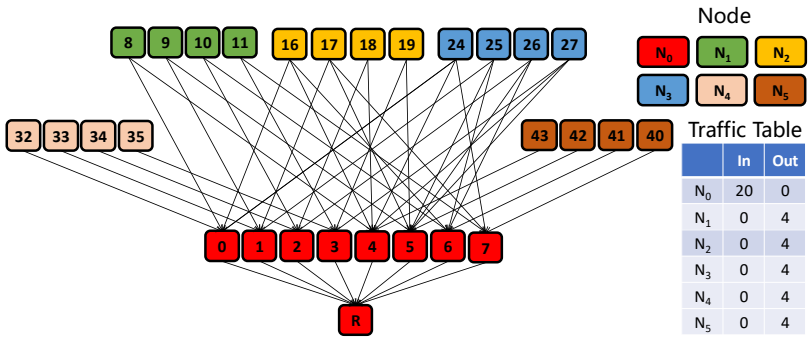
4.4 Application for Butterfly Codes

We now explain how the pECDAG can be applied to Butterfly codes [28] for parallel repair, using the $(6, 4)$ Butterfly code with $w = 8$ as an example. Each block is divided into $w = 8$ sub-blocks, and four uncoded blocks are encoded into six coded blocks (i.e., B_0, B_1, B_2, B_3, B_4, and B_5) stored across six independent nodes (i.e., N_0, N_1, N_2, N_3, N_4, and N_5). Suppose that we repair a lost block (say B_0) in a new node (say N_0).

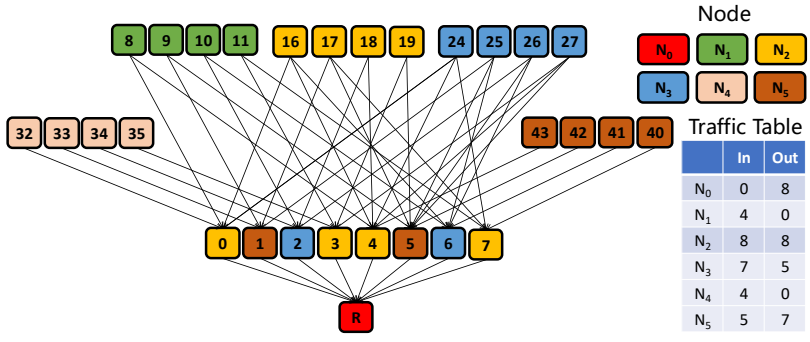
Figure 9(a) illustrates the pECDAG for the centralized repair of the $(6, 4)$ Butterfly code. N_0 downloads four sub-blocks (i.e., half of the block size) from other five nodes N_1, N_2, N_3, N_4, and N_5, resulting in 20 sub-blocks for both repair bandwidth and maximum repair load. Consequently, N_0 becomes the bottleneck. The affinity ratio for the conventional repair is $\frac{1}{9}$, as only R has affinity.

Figures 9(b) and 9(c) depict the parallel repair solutions generated by the pruning-based heuristic (§4.1) and the affinity-based heuristic (§4.3), respectively. The pruning-based heuristic reduces the maximum repair load to 8 sub-blocks, and increases the affinity ratio to $\frac{6}{9}$ compared to the centralized repair. The affinity-based heuristic reduces the maximum repair load to 9 sub-blocks, while increasing the affinity ratio to $\frac{8}{9}$ compared to the centralized repair.

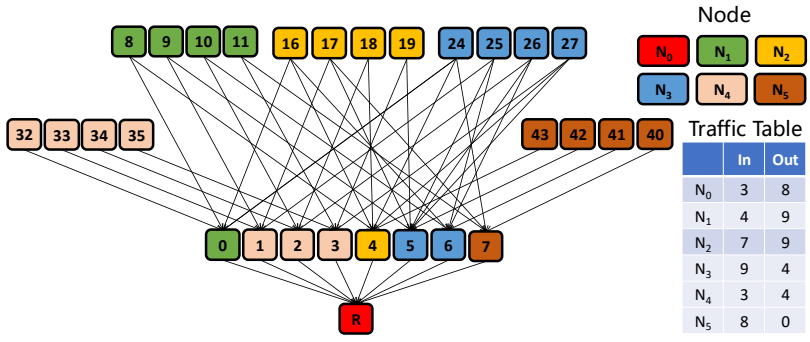
We note that increasing the affinity ratio does not always result in a lower maximum repair load for Butterfly codes (although it still reduces the maximum repair load compared to the centralized repair). The main reason is attributed to the structure of Butterfly codes. In the pECDAG for the (6,4) Butterfly code, each input vertex connects to multiple output vertices (i.e., v_{17} connects to v_1, v_5, and v_7), but these output vertices do not share many input vertices (i.e., v_1, v_5, and v_7 share v_{17}, but their other input vertices are distinct). Consequently, an input sub-block is often sent to multiple nodes to compute different sub-blocks, even though the affinity ratio is high. For example, in Figure 9(c), v_1, v_5, and v_7 all have affinity as each of them has the same color as one of its input vertices (e.g.,



(a) Centralized repair



(b) Pruning-based heuristic



(c) Affinity-based heuristic.

Fig. 9. Example of repairing B_0 using the $(6, 4)$ Butterfly code ($w = 8$) using the pECDAG.

v_{33}, v_{25}, and v_{40}, respectively). However, sub-block v_{17} needs to be sent to N_4 (pink-colored), N_3 (blue-colored), and N_5 (brown-colored) to compute sub-blocks v_1, v_5, and v_7, respectively, thereby incurring a high maximum repair load (even though the pECDAG has a high affinity ratio). In contrast, Clay codes typically have multiple output vertices sharing the same set of input vertices. For example, in Figure 8(a) (§4.3), v_{16} and v_{17} share input vertices v_9 and v_{12}. By assigning v_{16} and v_{17} the same color, we avoid sending v_9 and v_{12} to different nodes multiple times and keep a low maximum repair load. Nevertheless, both pruning-based and affinity-based heuristics generate parallel repair solutions with comparable repair bandwidth and maximum repair load, while the

affinity-based heuristic offers guaranteed algorithmic runtime. We evaluate the performance on Butterfly codes in §7.1.

4.5 Summary

We summarize the main insights of this section. The pruning-based heuristic (§4.1) aims to search for an efficient parallel repair solution, represented by a pECDAG, starting from a random solution to minimize repair bandwidth and maximum repair load. However, it neither explains why certain solutions achieve lower repair bandwidth and maximum repair load, nor offers runtime guarantees. Our affinity analysis (§4.2) reveals that parallel repair solutions with high affinity ratios generally exhibit lower repair bandwidth and maximum repair load, thereby providing insights into their performance. Building on the affinity analysis, the affinity-based heuristic (§4.3) generates repair solutions by assigning vertex colors based on input vertices to reduce repair bandwidth and maximum repair load. Although the affinity-based heuristic may slightly underperform the pruning-based heuristic for Butterfly codes (§4.4), its resulting repair solutions still have comparable repair bandwidth and maximum repair load to the pruning-based heuristic, while providing algorithmic runtime guarantees.

5 FULL-NODE RECOVERY

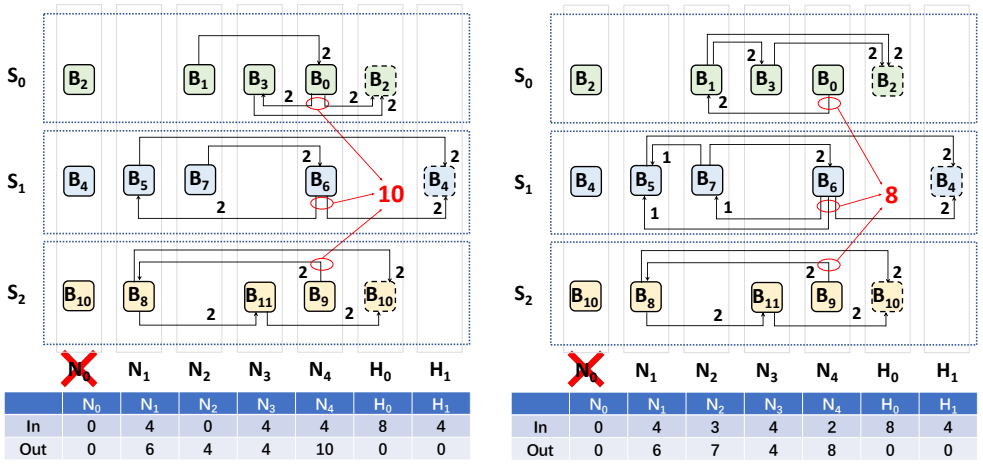
In this section, we extend our single-block repair design to full-node recovery, which repairs all lost blocks of a single failed node. Since the lost blocks span multiple stripes that are stored in different sets of nodes across the storage system, it is critical to effectively exploit system-wide parallelism for fast full-node recovery. We first demonstrate that our current single-block repair design, which focuses on *intra-stripe-only* parallel repair scheduling, still suffers from load imbalance in full-node recovery (§5.1). We then propose a co-design of intra-stripe and inter-stripe parallel repair scheduling, which schedules not only the parallel repair operations for sub-blocks within each lost block (i.e., intra-stripe parallel repair), but also the parallel repair operations across multiple lost blocks (i.e., inter-stripe parallel repair) (§5.2).

5.1 Limitations of Intra-Stripe-Only Parallel Repair

We consider full-node recovery in a *hot-standby* scenario, in which the storage system reserves a configurable number of hot-standby nodes that initially do not store any block. Full-node recovery repairs the lost blocks of a failed node and stores them on hot-standby nodes, which incur the highest repair load among all nodes. Thus, minimizing the maximum repair load across hot-standby nodes is critical for improving repair performance. Compared to the centralized repair of MSR codes, intra-stripe-only parallel repair only reduces the maximum repair load for repairing a single block (of a single stripe). However, it still causes load imbalance when repairing multiple blocks (across multiple stripes) in parallel. We depict via an example why intra-stripe-only parallel repair scheduling leads to load imbalance in full-node recovery.

Figure 10 shows the example. We consider a system with five active storage nodes N_0, N_1, N_2, N_3, and N_4, and two hot-standby nodes H_0 and H_1. We distribute three stripes s_0 (with blocks B_0, B_1, B_2, and B_3), s_1 (with blocks B_4, B_5, B_6, and B_7), and s_2 (with blocks B_8, B_9, B_{10}, and B_{11}) encoded under the $(4, 2)$ Clay code across different nodes in the system, as shown in Figure 10. Suppose that N_0 fails. We perform full-node recovery and distribute the repaired blocks (i.e., B_2, B_4, and B_{10}) across H_0 and H_1.

Figure 10(a) shows an example of intra-stripe-only parallel repair. We store the three repaired blocks in the hot-standby nodes in a round-robin manner, such that B_2 and B_{10} are repaired and stored in H_0, while B_4 is repaired and stored in H_1. We apply the affinity-based heuristic (§4) to schedule the repair for each block. We observe that in the repair of each lost block, the maximum



(a) Intra-stripe-only parallel repair (b) Intra-stripe and inter-stripe parallel repair

Fig. 10. Motivating example of intra-stripe and inter-stripe parallel repair scheduling for full-node recovery.

repair load is minimized to four sub-blocks (i.e., equivalent to the size of one block) based on the affinity-based heuristic. However, when we aggregate the repair of all three lost blocks, the aggregate maximum repair load becomes ten sub-blocks (in N_4), even exceeding the number of sub-blocks received by H_0. Load imbalance manifests in this example, as the repair operations of different blocks interfere with each other.

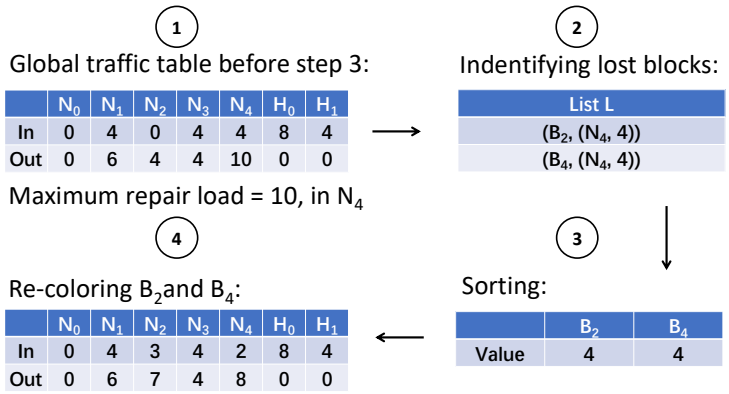
We argue that it is possible to apply different load allocations to the repair operations of different lost blocks, such that the interference among the repair operations can be mitigated. Figure 10(b) shows one such example, in which the load allocation for the repair of B_{10} is the same as in Figure 10(a), while the load allocations for the repair of B_2 and B_4 are different and N_4 only sends eight sub-blocks. Thus, the aggregate maximum repair load reduces to eight sub-blocks. This motivates us to explore a co-design for intra-stripe and inter-stripe parallel repair scheduling in full-node recovery.

5.2 Intra-Stripe and Inter-Stripe Parallel Repair

We now describe a co-design of intra-stripe and inter-stripe parallel repair for full-node recovery. Since the allocated resources for background routines are often capped [17, 44], instead of scheduling the repair of all lost blocks in the whole system, we partition all stripes in the entire system into *stripe groups*, each with a fixed number of stripes. We perform repair scheduling independently for each stripe group.

Our full-node recovery approach builds on the fast generation of intra-stripe parallel repair solutions from the affinity-based heuristic, so as to achieve effective inter-stripe parallel repair scheduling. At a high level, for each stripe group, it generates an initial repair solution for each lost block in the stripe group based on the affinity-based heuristic. It adjusts the repair solutions for some lost blocks to reduce the overall maximum repair load. The detailed steps to generate parallel repair scheduling for a stripe group are described as follows.

Step 1: Initialization. We first initialize an empty *global traffic table*, which records the aggregate incoming and outgoing traffic caused by the repair of all lost blocks in a stripe group. We will update the global traffic table in the following steps.



	N_0	N_1	N_2	N_3	N_4	H_0	H_1
In	0	4	0	4	4	8	4
Out	0	6	4	4	10	0	0

List L
(B_2, (N_4, 4))
(B_4, (N_4, 4))

	B_2	B_4
Value	4	4

	N_0	N_1	N_2	N_3	N_4	H_0	H_1
In	0	4	3	4	2	8	4
Out	0	6	7	4	8	0	0

Fig. 11. Example of inter-stripe repair scheduling for full-node recovery.

Step 2: Coloring. We perform intra-stripe repair scheduling. We first generate an initial repair solution for each of the lost blocks. For each initial repair solution to be generated for a lost block, we use the global traffic table that aggregates the repair traffic from the previously generated initial repair solutions as the input. Specifically, we first construct a pECDAG for the lost block by associating the colors with the intermediate vertices and the root vertex. For the intermediate vertices, we apply the affinity-based heuristic (§4) to generate the colors that reduce the aggregate maximum repair load. For the root vertex, we select a color that represents a hot-standby node selected from the existing hot-standby nodes in a round-robin manner.

Step 3: Optimization. We optimize the current full-node recovery solution in multiple iterations. In each iteration, we aim to identify the lost blocks for regenerating their repair solutions, such that the aggregate maximum repair load can be further reduced.

Step 3.1: Identifying lost blocks. We define a list L of records, each of which keeps a mapping of $(B, (N, b))$ (indexed by B) that specifies that repairing a lost block B will incur a repair bandwidth b to node N. We first compute the current aggregate maximum repair load (denoted by ℓ_{max}) based on the current repair solutions. We then identify the nodes whose incoming or outgoing bandwidth is equal to ℓ_{max} (note that multiple nodes can have the same incoming or outgoing bandwidth equal to ℓ_{max}). For each identified node (say N), we find the lost blocks whose repair bandwidth (denoted by b) contributes the most to ℓ_{max} of N (note that multiple lost blocks can contribute the same most repair bandwidth to the maximum repair load of N). For each such lost block (denoted by B), we add a record $(B, (N, b))$ to L, meaning that the repair of B incurs a repair bandwidth of b to N. Since the repair of B can also contribute the most repair bandwidth (say b') to another node (say N'), we append the mapping to the existing record, say $(B, (N, b), (N', b'))$.

Step 3.2: Re-coloring. We re-generate the repair solution based on L. For each lost block B in L, we sum the repair bandwidth incurred by all nodes in L, and sort the lost blocks in L by the sum of repair bandwidth in descending order. Then, for each lost block B in the sorted list L, we first remove its repair bandwidth from the global traffic table, and then re-color the corresponding pECDAG based on the affinity-based heuristic. If the new aggregate maximum repair load can be reduced after re-coloring, we repeat Step 3 for further possible optimization; otherwise, we stop the optimization.

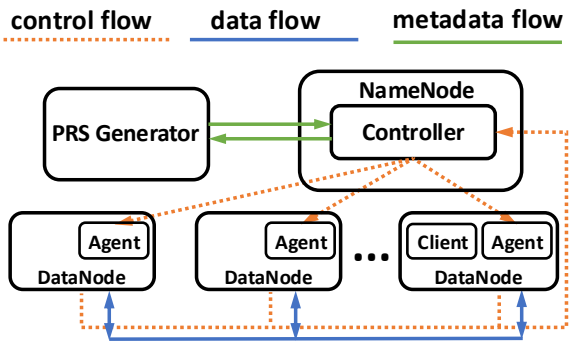


Fig. 12. System architecture.

Example. We follow the same example in Figure 10 to show inter-stripe repair scheduling in Step 3 can further reduce the aggregate maximum repair load. Figure 11 depicts the steps. We start with the global traffic table generated by the intra-stripe-only parallel repair shown in Figure 10(a).

In Step 3.1, we find that the maximum repair load is ten sub-blocks, located in N_4 (①). We analyze the repair bandwidth in N_4 and find that repairing B_2 and B_4 contributes four sub-blocks each to the maximum repair load in N_4. Thus, we update L with two records $(B_2, (N_4, 4))$, and $(B_4, (N_4, 4))$ (②). In this example, since the maximum repair load only occurs in N_4, each record in L only includes the repair bandwidth incurred by N_4.

In Step 3.2, we first find that the repair of B_2 and the repair of B_4 both have the sum of the repair bandwidth equal to four sub-blocks (③). We sort them by the sum of repair bandwidth in descending order. Suppose that the order is B_2 and B_4. We re-generate the repair solution for B_2, followed by B_4 (④). After the optimization, the maximum repair load reduces from ten to eight sub-blocks.

6 HyperParaRC DESIGN

We design and implement HyperParaRC, a parallel repair framework designed to balance repair bandwidth and maximum repair load for MSR-coded storage systems. We first describe the architecture of HyperParaRC (§6.1) and then elaborate on the implementation details of HyperParaRC (§6.2).

6.1 Architecture

We have built HyperParaRC based on OpenEC [23], an erasure coding framework that supports the deployment of custom erasure coding solutions in existing distributed storage systems. HyperParaRC runs as a middleware system and leverages OpenEC to deploy MSR codes on Hadoop HDFS [3]. HDFS stores data in fixed-size blocks and comprises a *NameNode* and multiple *DataNodes*: the NameNode manages the storage of all DataNodes and maintains the metadata of all stored blocks, while the DataNodes provide storage for the blocks. HyperParaRC performs encoding across HDFS blocks: for an (n, k) code, HyperParaRC encodes every k uncoded HDFS blocks (i.e., data blocks) into $n - k$ coded HDFS blocks (i.e., parity blocks) to form a stripe, which is stored in n DataNodes.

Figure 12 shows the architecture of HyperParaRC when it is integrated with HDFS. HyperParaRC includes a *PRS generator*, a *controller* that runs within the NameNode, and multiple *agents*, each of which runs within a DataNode. We also deploy a *client* that issues repair requests to HyperParaRC. We now elaborate on each component in detail.

PRS generator. The PRS generator computes parallel repair solutions for single-block repair and full-node recovery. For single-block repair, it pre-calculates a parallel repair solution based on the

affinity-based heuristic proposed in §4.3 and stores the results before the system starts [18]; this offline approach is suitable since the number of repair scenarios is limited for moderate ranges of (n, k) that are commonly used in practice [31]. When repairing a block, the PRS generator sends the parallel repair solution to the controller to coordinate the actual repair operations.

For full-node recovery, the PRS generator takes an online approach to generate parallel repair solutions for different stripe groups. It obtains the stripe metadata from the controller and generates parallel repair solutions for different stripe groups based on the co-design of intra-stripe and inter-stripe parallel repair scheduling proposed in §5.2. Each time the PRS generator forms a parallel repair solution for a stripe group, it sends the solution to the controller for actual repair operations and proceeds to generate a parallel repair solution for the next stripe group. This masks the time overhead of generating parallel repair solutions and ensures that the performance bottleneck lies in actual repair operations rather than solution generation.

Controller. The controller coordinates the parallel repair operations for the lost blocks encoded with MSR codes. For single-block repair, it reads the metadata of the block from HDFS to determine the location of other blocks in the same stripe. Then, the controller requests the parallel repair solution from the PRS generator. Finally, it translates the pECDAG into a set of *basic tasks* defined in OpenEC [23]. For full-node recovery, it reads the metadata of the lost blocks and divides them into stripe groups. For each stripe group, it requests parallel repair solutions from the PRS generator and translates the solution into basic tasks. There are four types of basic tasks: (i) reading sub-blocks from disk, (ii) fetching sub-blocks from other nodes, (iii) computing intermediate sub-blocks and repaired sub-blocks, and (iv) persisting the repaired sub-blocks as the final repaired blocks.

Agent. Agents are responsible for executing repair tasks generated by the controller to cooperatively repair lost blocks. Specifically, each agent reads locally stored blocks, performs erasure coding computations, and persists repaired blocks. An agent also sends or receives sub-blocks or intermediate sub-blocks via other agents. Thus, all the agents work together to repair blocks.

Client. A client sends repair requests to HyperParaRC. Specifically, it sends a request to the controller. Upon receiving a repair request, the controller starts to schedule the repair operation. The client waits until HyperParaRC finishes the repair operation. Note that a client can be co-located with an agent in a DataNode or run in a standalone machine outside of the DataNodes.

6.2 Implementation

We have implemented HyperParaRC in C++ with around 10 K LoC and integrated HyperParaRC into Hadoop-3.3.4 HDFS [3] (HDFS-3 for short). HyperParaRC uses Redis [8] for internal communications among the controller, agents, and clients. It uses Intel's Intelligent Storage Acceleration Library (ISA-L) [7] to perform encoding and decoding operations for erasure codes. It supports both centralized and parallel repair operations for MSR codes.

Implementation optimization for repair operations. To generate basic tasks for parallel repair, we need to carefully co-locate sub-block repair operations to avoid redundant data transmissions. For example, when deploying the pECDAG in Figure 4(a), we need to co-locate the repair of sub-blocks v_{18} and v_0, to ensure that the sub-blocks v_8 and v_{16} are only downloaded once in N_2 during the sub-block repair operation. To achieve this, we first divide the vertices into groups based on topological sorting, in which we can co-locate the sub-block repair operations for the vertices of the same color within the same group.

For example, the vertices in Figure 4(a) can be divided into the following five groups according to topological sorting: (i) v_4, v_5, v_8, v_9, v_{12}, and v_{13}; (ii) v_{16} and v_{17}; (iii) v_0, v_1, v_{18}, and v_{19}; (iv) v_2 and v_3; and (v) R. In group (ii), as v_{16} and v_{17} have the same color, we can co-locate the two sub-block repair operations, ensuring that N_2 only downloads sub-block v_{12} from N_3 once to compute the

two sub-blocks. Similarly, we can co-locate the sub-block repair operations specified by v_0 and v_{18} in N_2, and the sub-block repair operations specified by v_1 and v_{19} in N_0.

HDFS-3 integration. To improve parallelism, HyperParaRC divides the encoding of a stripe of blocks into the encoding of multiple small sub-stripes, where a data unit in each node of a sub-stripe is called a *packet*. In MSR codes, each packet contains w sub-packets. Each sub-stripe encodes $k \times w$ sub-packets into $n \times w$ MSR-coded sub-packets, where the size of a sub-packet is as small as 64 KiB. Thus, we implement sub-packetization across sub-packets instead of sub-blocks as in OpenEC [23], so that HyperParaRC can encode different sub-stripes in parallel to fully utilize system resources.

Note that HDFS-3 does not directly support MSR codes, so we rely on OpenEC to generate MSR-coded blocks and store them in HDFS-3. To enable parallel repair for MSR codes in HDFS-3, we run the HyperParaRC controller within the NameNode and run each HyperParaRC agent within a DataNode. The controller maintains a *stripe store* for MSR-coded stripes, which records the metadata of each stripe, including the blocks of the same stripe and the location of each block. We store the metadata of HDFS-3 blocks in the stripe store of HyperParaRC, allowing the controller to retrieve metadata from the stripe store when repairing a lost block.

Support for RS codes. In addition to MSR codes, HyperParaRC also supports RS codes. It implements both conventional centralized repair and parallel repair based on repair pipelining [24]. In repair pipelining, we divide a packet into sub-packets and pipeline the repair of different sub-packets across a repair path (i.e., each sub-packet is viewed as a slice in repair pipelining [24]). Note that RS codes have no sub-packetization and a sub-stripe encodes k packets into n RS-coded packets.

7 EVALUATION

We evaluate the performance of HyperParaRC on Alibaba Cloud [1]. The PRS generator operates on an `ecs.r7.2xlarge` instance with 8 vCPUs and 64 GiB RAM. The controller, agents (storage nodes), and hot-standby nodes operate on up to 25 `ecs.r7.xlarge` instances, each with 4 vCPUs and 32 GiB RAM (this setting includes up to 20 agents and 4 hot-standby nodes). Each instance is equipped with a 100 GiB enhanced SSD at performance level PL0 [2] and runs Ubuntu 18.04. All instances are interconnected via a 10 Gbps network. We aim to answer the following questions in our experiments:

- Q1: How does the affinity-based heuristic perform in finding the approximate MLP? (§7.1)
- Q2: How does the co-design of intra-stripe and inter-stripe parallel repair scheduling perform in finding the full-node recovery solutions? (§7.2)
- Q3: How does HyperParaRC perform under testbed experiments in terms of single-block repair and full-node recovery? (§7.3 and §7.4)
- Q4: What is the performance overhead of HyperParaRC when it runs atop HDFS-3 and how does it improve the repair performance of HDFS-3? (§7.5)

7.1 Performance of Finding the Approximate MLP

We start with single-block repair. We evaluate the algorithmic running time of finding the approximate MLP and examine how the resulting approximate MLP is related to the maximum repair load and the repair bandwidth.

(Exp#1) Algorithmic running time of finding the approximate MLP. We first examine how the affinity-based heuristic improves the running time performance of the pruning-based heuristic in finding an approximate MLP using the PRS generator (which runs on an `ecs.r7.2xlarge` instance). We compare the algorithmic running times of both heuristics with the brute-force approach, which can generate an exact MLP but only support small coding parameters.

(n, k, w)	**Brute-force**	**Pruning**	**Affinity**
$(4, 2, 4)$	264.1 s	1.8 s	0.81 ms
$(12, 8, 64)$	-	425.9 s	0.95 ms
$(14, 10, 256)$	-	57.2 h	345.8 ms
$(16, 12, 256)$	-	61.9 h	369.8 ms

(a) Clay codes

(n, k, w)	**Brute-force**	**Pruning**	**Affinity**
$(6, 4, 8)$	34.2 s	0.3 s	0.292 ms
$(12, 10, 512)$	-	31.64 h	0.7 s

(b) Butterfly codes

Table 3. (Exp#1) Algorithm running times of generating an MLP for single-block repair in the brute-force approach, the pruning-based heuristic (§4.1), and the affinity-based heuristic (§4.3). Note that both the pruning-based and affinity-based heuristics can only return an approximate MLP.

Table 3 shows the results for different combinations of (n, k) for Clay and Butterfly codes (note that the value of w is determined by n and k based on the codes). The brute-force approach can only find the MLP for the $(4, 2)$ Clay code and the $(6, 4)$ Butterfly code, but cannot return the solution for large (n, k) within reasonable time. Compared with the brute-force approach, both the pruning-based and affinity-based heuristics can generate an approximate MLP in a much shorter time. However, the pruning-based heuristic still takes a long time to search for an approximate MLP, while HyperParaRC reduces the running time to sub-seconds. For example, for the $(14, 10)$ Clay code, the pruning-based heuristic generates an approximate MLP in 57.2 h (i.e., over two days), while the affinity-based heuristic reduces the running time to 345.8 ms.

(Exp#2) Comparisons of maximum repair load and repair bandwidth. We compare the maximum repair load and repair bandwidth of five repair approaches: (i) the centralized repair for RS codes (RS); (ii) repair pipelining (RP) for RS codes [24]; (iii) the centralized repair for MSR codes (Clay or Butterfly) (§2.2); (iv) the parallel repair of MSR codes based on the pruning-based heuristic proposed in ParaRC [22] (§4.1) (ParaRC); and (v) the parallel repair of MSR codes based on the affinity-based heuristic proposed in §4.3 (HyperParaRC).

Table 4 shows the results for different coding parameters of Clay codes and Butterfly codes. HyperParaRC reduces the repair bandwidth of RS and RP, while reducing the maximum repair load of MSR codes. For example, for the $(14, 10)$ Clay code, HyperParaRC reduces the repair bandwidth of RS and RP from 10 blocks to 4.61 blocks, while achieving the minimum possible maximum repair load of 1 block. It also reduces the maximum repair load of the centralized repair of Clay codes from 3.25 blocks to 1 block, while incurring higher repair bandwidth. Similar trends are also observed for Butterfly codes.

Compared with ParaRC, HyperParaRC even reduces both the maximum repair load and the repair bandwidth for Clay codes under large (n, k) (i.e., $(12, 8)$, $(14, 10)$, and $(16, 12)$), while it has higher repair bandwidth than ParaRC with similar maximum repair loads for other cases. One possible reason for the improvements of HyperParaRC over ParaRC for Clay codes under large (n, k) is that there are a large number of intermediate vertices where HyperParaRC can adjust the colors to reduce both the maximum repair load and the repair bandwidth.

Thus, HyperParaRC can generate an approximate MLP that provides a better trade-off of the maximum repair load and the repair bandwidth than RS, RP, and the centralized repair for MSR codes, with significantly low algorithmic running time.

(n, k, w)	**RS**	**RP**	**Clay**	**ParaRC**	**HyperParaRC**
$(4, 2, 4)$	$(2, 2)$	$(1, 2)$	$(1.5, 1.5)$	$(1.25, 1.75)$	$(1, 2)$
$(12, 8, 64)$	$(8, 8)$	$(1, 8)$	$(2.75, 2.75)$	$(1.03, 4.78)$	$(1, 3.25)$
$(14, 10, 256)$	$(10, 10)$	$(1, 10)$	$(3.25, 3.25)$	$(1.06, 6.29)$	$(1, 4.61)$
$(16, 12, 256)$	$(12, 12)$	$(1, 12)$	$(3.75, 3.75)$	$(1.09, 6.93)$	$(1, 5.21)$

(a) Clay codes

(n, k, w)	**RS**	**RP**	**Butterfly**	**ParaRC**	**HyperParaRC**
$(6, 4, 8)$	$(4, 4)$	$(1, 4)$	$(2.5, 2.5)$	$(1, 3.5)$	$(1.13, 4.25)$
$(12, 10, 512)$	$(10, 10)$	$(1, 10)$	$(4.09, 4.09)$	$(1.16, 8.66)$	$(1.55, 11.72)$

(b) Butterfly codes

Table 4. (Exp#2) Comparisons of (maximum repair load, repair bandwidth) of different repair approaches, measured in terms of the number of blocks.

We observe that the reduction in maximum repair load and repair bandwidth varies across different (n, k) for the same MSR code and across different MSR codes. This variation is due to the distinct graph structures of pECDAGs for different parameters and codes, and their pECDAGs differ in the numbers of vertices and edges. Consequently, the improvements from parallel repair scheduling also vary. Given that HyperParaRC's performance depends on the choices of MSR codes and coding parameters, we recommend initially deploying the affinity-based heuristic in real systems, while also running the pruning-based heuristic offline. The parallel repair solution returning the lowest repair bandwidth and maximum repair load is then applied.

7.2 Simulations for Full-node Recovery

We consider full-node recovery under large-scale simulations. We evaluate the performance of the co-design of intra-stripe and inter-stripe parallel repair scheduling (§5.2) in terms of the (aggregate) maximum repair load, the (aggregate) repair bandwidth, and the algorithmic running time to generate repair solutions for all lost blocks. We run our simulations on the `ecs.r7.2xlarge` instance (which hosts the PRS generator).

In our simulations, we consider a cluster of 100 storage nodes. The cluster is configured with 1,000 stripes that are randomly distributed across storage nodes. We ensure that each stripe contains a block stored in a pre-selected storage node that will later be treated as a failed node, so as to allow for the repair of 1,000 blocks in full-node recovery. By default, we focus on the (14,10) Clay code and recover the lost blocks in four hot-standby repair nodes (in addition to the 100 storage nodes). We configure the stripe group size as four stripes and distribute the four repaired blocks of each stripe group across the four hot-standby nodes. We plot the average results over five runs, including the error bars showing the 95% confidence interval based on the Student's t-distribution.

We compare six repair approaches for full-node recovery, including RS, RP, the centralized repair for MSR codes, ParaRC, HyperParaRC-base, and HyperParaRC. Note that for the first four approaches, each block is repaired as described in Exp#2 (§7.1), while the repaired block is placed across hot-standby nodes in a round-robin manner. For HyperParaRC-base, each block is repaired by the approximate MLP generated by the affinity-based heuristic without inter-stripe parallel repair scheduling. HyperParaRC is the complete version that applies intra-stripe and inter-stripe parallel repair scheduling for full-node recovery.

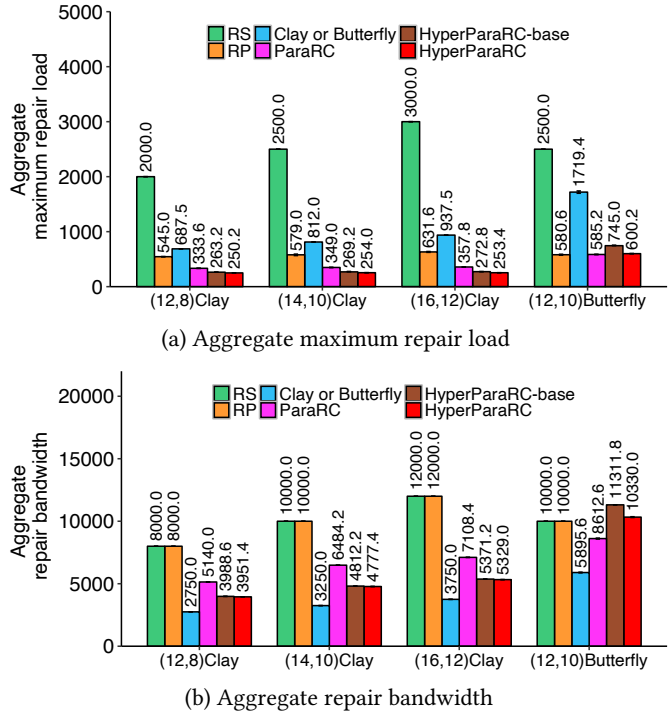


Fig. 13. (Exp#3) Impact of MSR codes on full-node recovery in simulations.

(Exp#3) Impact of MSR codes on full-node recovery in simulations. We focus on the (12, 8) Clay code, the (14, 10) Clay code, the (16, 12) Clay code, and the (12, 10) Butterfly code to evaluate the maximum repair load and the repair bandwidth. Figure 13 shows the results.

For Clay codes, HyperParaRC-base significantly reduces the aggregated maximum repair load compared with RS, RP, Clay, and ParaRC. Applying intra-stripe and inter-stripe parallel repair scheduling, HyperParaRC further slightly reduces the aggregated maximum repair load compared with HyperParaRC-base. For example, for the (16, 12) Clay code, HyperParaRC-base reduces the maximum repair load by 90.9%, 56.8%, 70.9%, and 23.8% compared with RS, RP, Clay, and ParaRC, respectively. HyperParaRC slightly reduces the maximum repair load by 7.1% compared with HyperParaRC-base. Both HyperParaRC-base and HyperParaRC have much less repair bandwidth than RS, RP, and ParaRC, while HyperParaRC only slightly reduces the repair bandwidth compared with HyperParaRC-base. For example, for the (16, 12) Clay code, HyperParaRC-base reduces the repair bandwidth by 55.2%, 55.2%, and 24.4% compared with RS, RP, and ParaRC, respectively. Note that the repair bandwidth of HyperParaRC cannot be less than that of Clay codes, which theoretically minimize the repair bandwidth.

For Butterfly codes, we observe different findings. For the (12, 10) Butterfly code, HyperParaRC-base reduces the maximum repair load by 70.2% and 56.7% compared with RS and Butterfly, respectively. However, the maximum repair load of HyperParaRC-base is larger than those of RP and ParaRC. The reason is that HyperParaRC-base repairs each block based on the affinity-based heuristic without inter-stripe parallel repair scheduling. As shown in §7.1, the affinity-based heuristic for Butterfly codes, while reducing the algorithmic running time, can incur high

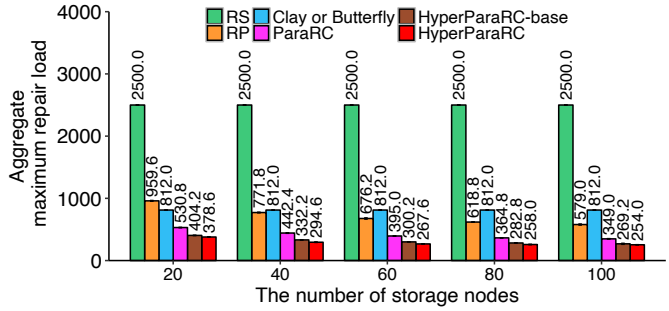


Fig. 14. (Exp#4) Impact of the number of nodes on full-node recovery in simulations.

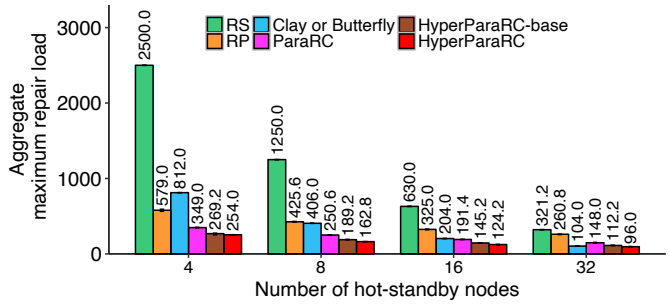


Fig. 15. (Exp#5) Impact of the number of hot-standby nodes on full-node recovery in simulations.

maximum repair load and repair bandwidth. Nevertheless, after applying intra-stripe and inter-stripe parallel repair scheduling, HyperParaRC reduces the maximum repair load by 19.4% and the repair bandwidth by 8.6% compared with HyperParaRC-base, thereby showing the significance of intra-stripe and inter-stripe parallel repair scheduling.

(Exp#4) Impact of the number of nodes on full-node recovery in simulations. We study the performance of HyperParaRC when the number of nodes is varied. Figure 14 shows the results. The maximum repair load of RS and Clay is not influenced by the number of nodes, as the bottleneck lies in the hot-standby node. For RP, ParaRC, HyperParaRC-base, and HyperParaRC, the maximum repair load decreases as the number of nodes increases. Among all approaches, HyperParaRC incurs the smallest maximum repair load, benefiting from the intra-stripe and inter-stripe parallel repair scheduling. For example, when there are 20 nodes, HyperParaRC-base reduces the maximum repair load by 83.8%, 57.8%, 50.2%, and 23.9%, compared with RS, RP, Clay, and ParaRC, respectively. HyperParaRC further reduces the maximum repair load by 6.3% compared with HyperParaRC-base.

(Exp#5) Impact of the number of hot-standby nodes on full-node recovery in simulations. We study the performance of HyperParaRC when the number of hot-standby nodes is varied. We configure the number of stripes in a stripe group to be the same as the number of hot-standby nodes. Figure 15 shows the results. Increasing the number of hot-standby nodes leads to less aggregate maximum repair load by distributing the repaired blocks across hot-standby nodes. With the same number of hot-standby nodes, HyperParaRC always has the least maximum repair load among all approaches. For example, when there are 16 hot-standby nodes, HyperParaRC reduces the maximum repair load by 80.3%, 61.8%, 39.1%, 35.1%, and 14.5% compared with RS, RP, Clay, ParaRC, and HyperParaRC-base, respectively.

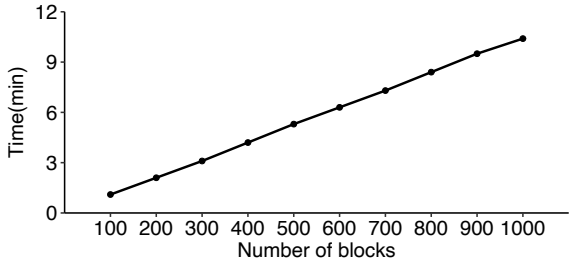


Fig. 16. (Exp#6) Algorithmic running time of finding full-node recovery solutions.

(Exp#6) Algorithmic running time of finding full-node recovery solutions. We evaluate the algorithmic running time of HyperParaRC in finding full-node recovery solutions by varying the number of lost blocks to be repaired in full-node recovery from 100 to 1,000; note that the time does not include the actual repair operations of reconstructing lost blocks. Figure 16 shows the results. The algorithmic running time increases linearly with the number of blocks being repaired. HyperParaRC only needs 10.4 minutes to generate a full-node recovery solution for 1,000 blocks. In real deployment, the algorithmic running time is overlapped with the actual repair operations (§6.2), so solution generation is not the bottleneck in full-node recovery.

7.3 Testbed Experiments for Single-Block Repair

We now study the single-block repair performance via testbed experiments on Alibaba Cloud. By default, we focus on the $(14, 10)$ Clay code, with the block size 256 MiB and the sub-packet size 64 KiB. In this case, the packet size is 256×64 KiB = 16 MiB, meaning that a stripe can be divided into 16 sub-stripes. We compare the five single-block repair approaches as described in Exp#2 (§7.1). We report the average single-block repair time for the k original uncoded blocks, with error bars showing the 95% confidence interval based on the Student's t-distribution.

(Exp#7) Impact of MSR codes on single-block repair in the cloud. We evaluate the single-block repair performance of HyperParaRC for different MSR code configurations, including the $(12, 8)$ Clay code, the $(14, 10)$ Clay code, the $(16, 12)$ Clay code, and the $(12, 10)$ Butterfly code.

Figure 17 shows the evaluation results. The results are consistent with our findings in Exp#2 (see our explanations in Exp#2 in §7.1): in all cases, HyperParaRC outperforms RS, RP, and the centralized repair for both Clay and Butterfly codes. It outperforms ParaRC for Clay codes, and underperforms for Butterfly codes. For example, for the $(16, 12)$ Clay code, HyperParaRC reduces the single-block repair time by 81.6%, 62.5%, 68.2%, and 21.3% compared with RS, RP, Clay, and ParaRC, respectively. Note that even though RP minimizes the maximum repair load, its single-block repair time is not necessarily minimized as it still has high repair bandwidth and needs to read the whole block from each available node. For the $(12, 10)$ Butterfly code, HyperParaRC reduces the single-block repair time by 37.7% and 16.8% compared with RS and Clay, respectively, while it is 10.7% slower than that of ParaRC.

(Exp#8) Impact of sub-packet size on single-block repair in the cloud. We evaluate the single-block repair time of HyperParaRC under different sub-packet sizes. We vary the sub-packet size from 16 KiB to 256 KiB, and fix the block size at 256 MiB. Note that the packet size is the sub-packet size multiplied by w, where w depends on the erasure code.

Figure 18 shows the results. HyperParaRC has the least single-block repair time compared with other repair approaches for all sub-packet sizes. For example, when the sub-packet size is 128 KiB, HyperParaRC reduces the single-block repair time by 72.7%, 53.5%, 56.9%, and 7.8% compared

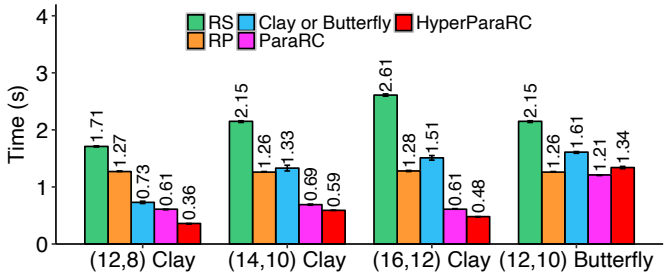


Fig. 17. (Exp#7) Impact of MSR codes on single-block repair in the cloud.

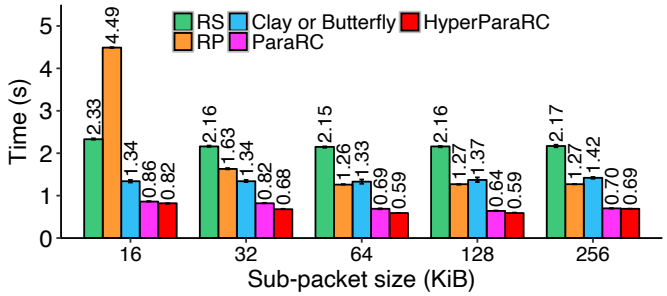


Fig. 18. (Exp#8) Impact of sub-packet size on single-block repair in the cloud.

with RS, RP, Clay, and ParaRC, respectively. We observe that the performance HyperParaRC drops when the sub-packet size is reduced to 16 KiB due to the overhead of processing a large number of sub-packets. For example, when the sub-packet size reduces from 64 KiB to 16 KiB, the single-block repair time of HyperParaRC increases from 0.59 s to 0.82 s. Also, we observe that when the sub-packet size increases to 256 KiB, the single-block repair time increases to 0.69 KiB, as a larger sub-packet size implies that a block is divided into fewer packets and the repair parallelism diminishes.

(Exp#9) Impact of block size on single-block repair in the cloud. We evaluate HyperParaRC under different block sizes. We vary the block size from 64 MiB to 512 MiB, and fix the sub-packet size at 64 KiB. The block size determines the number of sub-stripes. For example, for the default block size of 256 MiB, the number of sub-stripes for the (14,10) Clay code is 16.

Figure 19 shows the results. HyperParaRC outperforms other repair approaches when the block size is sufficiently large (e.g., at least 128 MiB). For example, when the block size is 512 MiB, HyperParaRC reduces the single-block repair time by 75.6%, 70.8%, 49.3%, and 21.1% compared with RS, RP, Clay, and ParaRC respectively. When the block size is small, RP outperforms parallel repair of MSR codes. For example, when the block size is 64 MiB, HyperParaRC has 23.8% higher single-block repair time than RP. The reason is that when the block size is small, the parallel repair of Clay codes suffers from the overhead of high sub-packetization. For example, when the block size is 64 MiB, a stripe can only be divided into four sub-stripes, so the degree of parallelism is low. In contrast, RP can pipeline the repair of 1,024 sub-stripes (§6.2) and outperform ParaRC and HyperParaRC for small block sizes. Thus, HyperParaRC is suitable for the deployment scenarios with large block sizes, which are commonly found in modern distributed storage systems (§2.1).

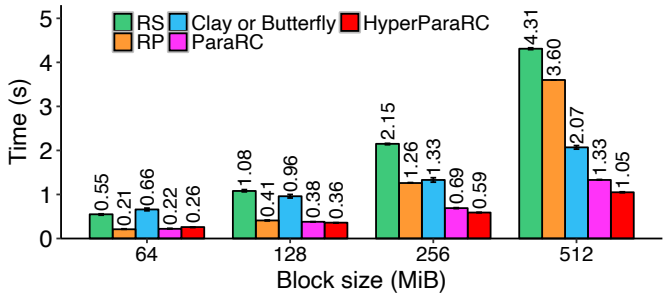


Fig. 19. (Exp#9) Impact of block size on single-block repair in the cloud

7.4 Testbed Experiments for Full-Node Recovery

We evaluate the performance of full-node recovery in a real network environment on Alibaba Cloud. We measure the total time of repairing 20 lost blocks of a failed storage node from 20 stripes, whose available blocks are randomly distributed across the non-failed storage nodes. By default, we configure four hot-standby nodes and the stripe group size as four stripes. We follow the same system settings as described in §7.3. We compare the six full-node recovery approaches as described in §7.2. We report the average full-node recovery time over five runs, with error bars showing the 95% confidence intervals based on the Student's t-distribution.

(Exp#10) Impact of network bandwidth on full-node recovery in the cloud. We first evaluate the performance of HyperParaRC under 1 Gbps and 10 Gbps. We consider the 1 Gbps setting [37] to address a bandwidth-throttled scenario [17, 44] and configure the network bandwidth using the Wondershaper tool [16].

Figure 20 shows the results. We first compare the RS-based approaches (i.e., RS and RP) with the MSR-based approaches (i.e., Clay, ParaRC, HyperParaRC-base, and HyperParaRC). In both bandwidth settings, the MSR-based approaches achieve higher full-node recovery performance than the RS-based approaches, as the RS-based approaches incur much higher aggregate repair bandwidth (even though RP has less aggregate maximum repair load). The poor performance of the RS-based approaches is more pronounced in the 1 Gbps setting.

We next compare Clay and ParaRC. Different from our observations in single-block repair, ParaRC performs worse than Clay in full-node recovery. For example, ParaRC is 8.4% and 31.2% slower than Clay in the 1 Gbps and 10 Gbps settings, respectively. There are two reasons. First, in full-node recovery, Clay also benefits from repairing multiple blocks in parallel in different hot-standby nodes. Second, even though ParaRC has less aggregate maximum repair load than Clay, its aggregate repair bandwidth is much higher than that of Clay (see Exp#3).

We further compare HyperParaRC-base with Clay and ParaRC. HyperParaRC-base outperforms ParaRC as it benefits from both lower aggregate maximum repair load and lower aggregate repair bandwidth through the affinity-based heuristic (see Exp#3). For example, HyperParaRC-base is 47.5% and 18.0% faster than ParaRC in the 1 Gbps and 10 Gbps settings, respectively. However, HyperParaRC-base still cannot guarantee improved performance over Clay for full-node recovery. For example, HyperParaRC-base is 43.1% faster than Clay in the 1 Gbps setting, but is 7.5% slower than Clay in the 10 Gbps setting.

Finally, HyperParaRC achieves the best full-node recovery performance through intra-stripe and inter-stripe parallel repair scheduling in both network settings by further reducing both the maximum repair load and repair bandwidth of HyperParaRC-base (see Exp#3). For example, in the

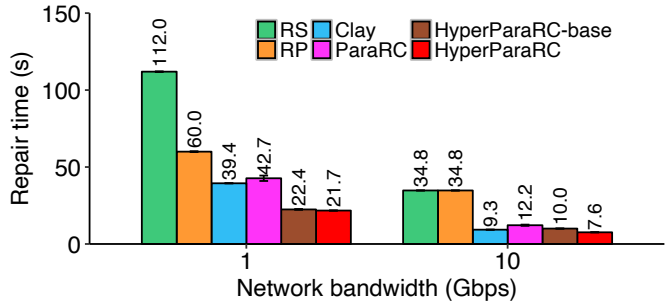


Fig. 20. (Exp#10) Impact of network bandwidth on full-node recovery in the cloud.

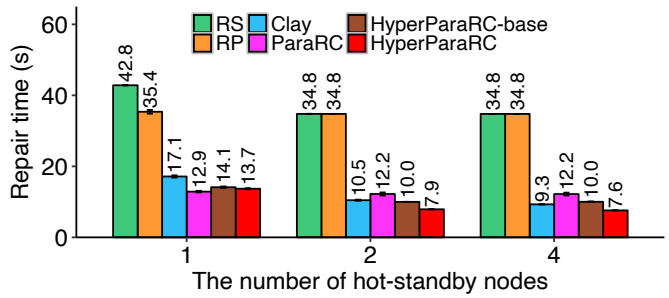


Fig. 21. (Exp#11) Impact of the number of hot-standby nodes on full-node recovery in the cloud.

1 Gbps setting, HyperParaRC is 44.9%, 49.2%, and 3.1% faster than Clay, ParaRC, and HyperParaRC-base, respectively, and in the 10 Gbps setting, it is 18.3%, 37.7%, and 24.0% faster, respectively.

(Exp#11) Impact of the number of hot-standby nodes on full-node recovery in the cloud. We evaluate the full-node recovery time of HyperParaRC by varying the number of hot-standby nodes as one, two, and four. Figure 21 shows the results. When there is only one hot-standby node, both HyperParaRC and ParaRC significantly outperform RS, RP, and Clay. For example, HyperParaRC is 67.9%, 61.3%, and 19.9% faster than RS, RP, and Clay, respectively. Note that both HyperParaRC and ParaRC show similar performance, the only hot-standby node is the bottleneck. When the number of hot-standby nodes increases, we observe limited performance improvements of RS and RP, as they incur high repair bandwidth and their bottleneck lies in the agents, which read and send data for full-node recovery, instead of in hot-standby nodes. For Clay, ParaRC, HyperParaRC-base, and HyperParaRC, increasing the number of hot-standby nodes reduces the recovery time. Among all approaches, HyperParaRC achieves the best full-node recovery performance. For example, when there are two hot-standby nodes, HyperParaRC is 77.3%, 77.3%, 24.8%, 35.3%, and 21.0% faster than RS, RP, Clay, ParaRC, and HyperParaRC-base, respectively.

(Exp#12) Impact of MSR codes on full-node recovery in the cloud. We evaluate HyperParaRC by considering different MSR code configurations, including the $(12, 8)$ Clay code, the $(14, 10)$ Clay code, the $(16, 12)$ Clay code, and the $(12, 10)$ Butterfly code. Figure 22 shows the results. We observe significant performance improvements of HyperParaRC for Clay codes. For example, for the $(16, 12)$ Clay code, HyperParaRC-base is 71.2%, 71.2%, 5.2%, and 12.1% faster than RS, RP, Clay, and ParaRC, respectively, and when applying intra-stripe and inter-stripe parallel repair scheduling, HyperParaRC is 24.8% faster than HyperParaRC-base.

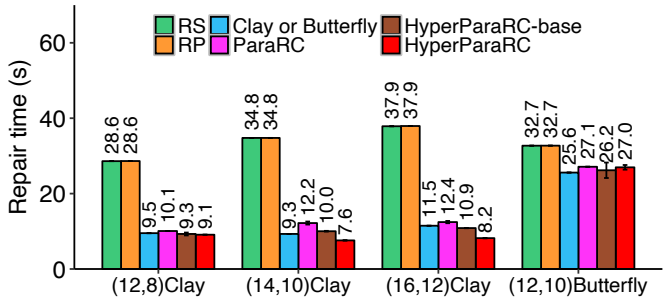


Fig. 22. (Exp#12) Impact of MSR codes on full-node recovery in the cloud.

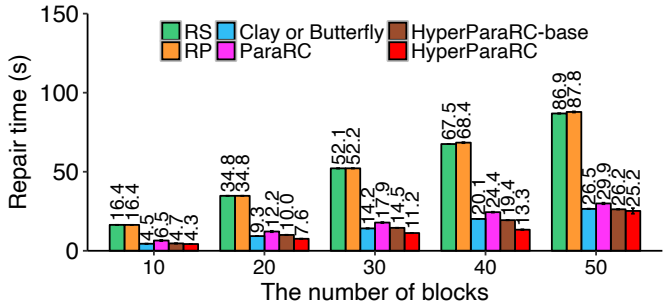


Fig. 23. (Exp#13) Impact of the number of repaired blocks on full-node recovery in the cloud.

However, for the (12, 10) Butterfly code, we find that the parallel repair for Butterfly codes is not necessarily faster than the centralized repair of Butterfly codes in full-node recovery. There are two reasons. First, even though the parallel repair reduces the maximum repair load, the repair bandwidth of ParaRC, HyperParaRC-base, and HyperParaRC remains much higher than that of Butterfly (see Exp#3). Second, the (12, 10) Butterfly code has a high sub-packetization level, in which each packet is divided into 512 sub-packets and hence the packet size is 512×64 KiB = 32 MiB. Thus, the parallel repair has marginal benefits.

(Exp#13) Impact of the number of repaired blocks on full-node recovery in the cloud. We evaluate HyperParaRC when it repairs different numbers of blocks, varying from 10 to 50. Figure 23 shows the results. When the number of repaired blocks increases, the overall recovery time increases for all approaches and HyperParaRC still achieves the best performance. For example, when repairing 40 blocks, HyperParaRC is 80.3%, 80.6%, 33.8%, 45.5%, and 31.4% faster than RS, RP, Clay, ParaRC, and HyperParaRC-base, respectively.

(Exp#14) Impact of the stripe group size. We study the impact of stripe group size. We repair 128 blocks and vary the group size to 4, 8, and 16. Figure 24 shows the results. When the stripe group size is 4, HyperParaRC reduces the repair time by 44.8%, 45.7%, 19.4%, 23.7%, and 14.9% compared to RS, RP, Clay, ParaRC, and HyperParaRC-base, respectively. However, when the stripe group size increases, we observe that HyperParaRC's improvements diminish. The reason is that when the group size is 4, HyperParaRC's intra-stripe and inter-stripe repair scheduling balances repair traffic across four hot-standby nodes, and adding more stripes per group causes significant resource contention across all nodes. In contrast, other repair methods benefit from larger group sizes, as they can leverage underutilized node resources to (slightly) reduce repair time. Nevertheless, HyperParaRC consistently achieves the shortest repair time among all approaches.

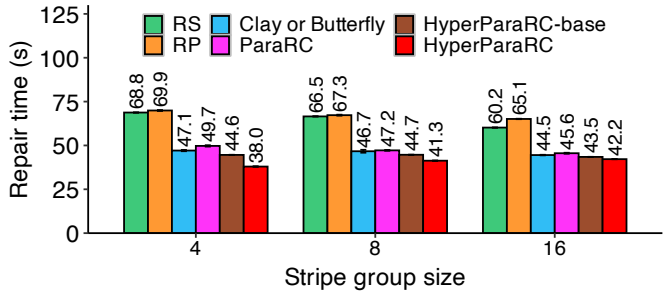


Fig. 24. (Exp#14) Impact of the stripe group size on full-node recovery in the cloud.

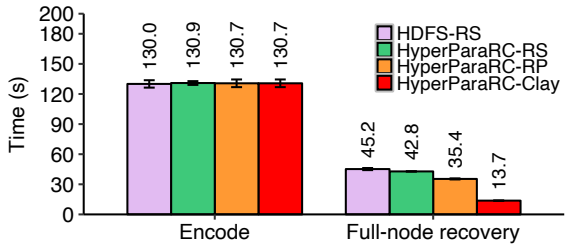


Fig. 25. (Exp#15) HDFS-3 integration.

7.5 Performance of HDFS-3 Integration

(Exp#15) HDFS-3 integration. We evaluate the integration of HyperParaRC into HDFS-3. Recall that we have shown the benefits of HyperParaRC over other repair approaches in §7.3 and §7.4. In this experiment, we only focus on the performance overhead and performance gain of HyperParaRC in HDFS-3 deployment. We focus on the (14, 10) Clay code with the default block size of 256 MiB.

Currently, HDFS-3 does not support Clay codes in its codebase, so we mainly compare HyperParaRC with the RS code implementation in HDFS-3. For encoding, we evaluate the overhead of encoding 20 stripes. For full-node recovery, we evaluate the overhead of repairing 20 lost blocks of a failed node with only one hot-standby node. We omit the results for single-block repair, as HDFS-3 does not trigger the repair of a single block. Even if HDFS-3 issues a degraded read of a file under node failures, it always returns all blocks of the original file to the client. Thus, in this case, the centralized repair of the lost block is sufficient.

We consider four approaches: (i) encoding by the default RS codes and performing the default recovery approach in HDFS (denoted by HDFS-RS); (ii) encoding by RS codes and performing the centralized repair for RS codes in HyperParaRC (denoted by HyperParaRC-RS); (iii) encoding by RS codes and performing repair pipelining [24] in HyperParaRC (denoted by HyperParaRC-RP); and (iv) encoding by Clay codes and performing the parallel repair via intra-stripe and inter-stripe parallel repair in HyperParaRC (denoted by HyperParaRC-Clay).

Figure 25 shows the results. For encoding, the encoding of Clay codes in HyperParaRC and the encoding of RS codes in HDFS-3 have similar overhead. For example, HDFS-RS takes 130.0 s, while HyperParaRC-Clay takes 130.7 s for encoding 20 stripes. For full-node recovery, HyperParaRC-Clay reduces the full-node recovery time by 70.9% compared with HDFS-RS.

8 RELATED WORK

RS codes [36] are popularly deployed in distributed storage systems [4--6, 11, 27, 30], but incur high repair bandwidth (§2.1). Thus, research efforts are made to improve the repair performance of RS codes. One direction is to design fast repair algorithms over RS codes, while another direction is to design regenerating codes to minimize the repair bandwidth.

Repair algorithms for RS codes. PPR [26] divides the repair of a block into partial operations and parallelizes them for improved repair performance. Repair pipelining [21, 24] divides the repair of a block into the repair of small slices, organizes the available nodes that participate in the repair operation into a repair path, and pipelines the repair of slices along the repair path to reduce the degraded read time to be almost the same as the time of reading a block. PPT [9], SMFRepair [48], and PivotRepair [47] propose different parallel repair strategies for RS codes in heterogeneous bandwidth environments. However, the above repair algorithms do not reduce the repair bandwidth of RS codes. Our work focuses on designing parallel repair algorithms for regenerating codes, which have much lower repair bandwidth than RS codes.

Regenerating codes. Regenerating codes [10] are a family of erasure codes that minimize the repair bandwidth. Minimum-storage regenerating (MSR) codes not only minimize the repair bandwidth, but also achieve the MDS property. Many research studies propose new designs of MSR codes, including F-MSR codes [13], PM-RBT codes [32], Butterfly codes [28], and Clay codes [43]. Such MSR codes operate in different parameter regimes, such as $n - k = 2$ for F-MSR codes [13] and Butterfly codes [28], and $n \geq 2k - 1$ for PM-RBT codes [32]. In particular, Clay codes [43] are state-of-the-art MSR codes that support general parameters of n and k and are proven to minimize both repair bandwidth and I/Os (§1). Geometric partitioning [40] builds on Clay codes and divides an object into variable-sized blocks to trade between the performance of degraded reads and full-node recovery. However, the repair strategy for existing MSR codes is still based on the centralized repair approach, in which a node downloads the required data from all available nodes to repair a failed block. Even though the repair bandwidth is still the minimum, the maximum repair load is high. ParaRC addresses this trade-off by proposing a parallel repair strategy for MSR codes.

DAG-based erasure coding. OpenEC [23] proposes an ECDAG abstraction to model and configure erasure coding operations as a directed acyclic graph (DAG) without modifying the I/O workflows of the underlying distributed storage system. RepairBoost [25] proposes a DAG abstraction to load-balance the full-node recovery workflow. Our work extends ECDAG [23] to support the parallel repair for MSR codes.

Full-node recovery. FastPR [41] improves the performance of repairing a soon-to-fail node in RS-coded storage. It repairs blocks in parallel by either migration or erasure-coding-based reconstruction. Some studies, such as SelectiveEC [46] and RepairBoost [25], explore parallelization for full-node recovery based on RS codes. Shan et al. [39] explore block placement for load-balanced full-node recovery. To our knowledge, HyperParaRC is the first work that studies the scheduling of full-node recovery for MSR codes. It proposes a co-design of intra-stripe and inter-stripe parallel repair scheduling for full-node recovery.

9 DISCUSSION

Small block sizes. Although this work focuses on systems with large block sizes, HyperParaRC's parallel repair technique is applicable to systems with small block sizes. A pre-requisite is the use of MSR codes with low sub-packetization, as high sub-packetization MSR codes incur significant I/O overhead in systems with small block sizes. For example, some MSR codes (e.g., HashTag [20], NCBlob [12]) are designed with low sub-packetization and hence can be considered to leverage HyperParaRC for repair parallelization.

Scattered full-node recovery. We currently focus on hot-standby full-node recovery, where the hot-standby repair nodes are the bottleneck in the recovery process. HyperParaRC improves performance over existing repair approaches through intra-stripe and inter-stripe parallel repair scheduling. For *scattered* full-node recovery, where each node can serve as a repair node that stores the repaired block, HyperParaRC still relies on intra-stripe and inter-stripe parallel repair scheduling for load balancing. However, in scattered full-node recovery, the centralized repair for MSR codes may benefit from inter-stripe parallel repair scheduling by storing the repaired block of each stripe across all available nodes. The comparisons for hot-standby and scattered full-node recovery are posed as future work.

10 CONCLUSIONS AND FUTURE WORK

We present HyperParaRC, a parallel repair framework that aims to improve the repair performance of MSR-coded distributed storage systems. We show that there is a trade-off between repair bandwidth and maximum repair load in MSR codes. HyperParaRC exploits the sub-packetization nature of MSR codes by parallelizing the repair at the sub-block granularity. It builds on an affinity-based heuristic to minimize the maximum repair load, while maintaining the low repair bandwidth in polynomial time. It further adopts a co-design of intra-stripe and inter-stripe parallel repair scheduling for full-node recovery. We implement HyperParaRC that runs atop HDFS and evaluate it on Alibaba Cloud. Our evaluation results demonstrate the performance improvements of HyperParaRC in both single-block repair and full-node recovery compared with several state-of-the-line baselines, including our previously proposed ParaRC in the conference version.

We discuss possible future research directions. HyperParaRC currently focuses on the parallel repair of MSR codes in homogeneous network settings. One possible extension is to address the heterogeneous network settings with varying available bandwidth across different nodes and links. Another possible extension is to address scattered full-node recovery in addition to hot-standby full-node recovery (§9). Also, HyperParaRC focuses on optimizing the repair of a single failed block in a stripe, while optimizing the repair of multiple failed blocks in a stripe is not considered. The latter case is of particular interest for wide stripes [14, 17], where concurrently failed blocks in a stripe become more prevalent.

REFERENCES

[1] Accessed in Sept. 2025. Alibaba Cloud - Elastic Compute Service. https://www.alibabacloud.com/product/ecs-pricing-list/en.

[2] Accessed in Sept. 2025. Alibaba Cloud - ESSDs. https://www.alibabacloud.com/help/en/elastic-compute-service/latest/essds.

[3] Accessed in Sept. 2025. Apache Hadoop 3.3.4 - HDFS Architecture. https://hadoop.apache.org/docs/r3.3.4/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[4] Accessed in Sept. 2025. Apache Hadoop 3.3.4 - HDFS Erasure Coding. https://hadoop.apache.org/docs/r3.3.4/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[5] Accessed in Sept. 2025. Backblaze Vaults: Zettabyte-Scale Cloud Storage Architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/.

[6] Accessed in Sept. 2025. Ceph - Erasure code. http://docs.ceph.com/docs/master/rados/operations/erasure-code/.

[7] Accessed in Sept. 2025. Intel Intelligent Storage Acceleration Library. https://github.com/intel/isa-l.

[8] Accessed in Sept. 2025. redis.io. https://redis.io/.

[9] Yunren Bai, Zihan Xu, Haixia Wang, and Dongsheng Wang. 2019. Fast Recovery Techniques for Erasure-Coded Clusters in Non-uniform Traffic Network. In *Proceedings of the 48th International Conference on Parallel Processing*.

[10] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. 2010. Network Coding for Distributed Storage Systems. *IEEE Transactions on Information Theory* 56, 9 (2010), 4539--4551.

[11] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in Globally Distributed Storage Systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*.

[12] Chuang Gan, Yuchong Hu, Leyan Zhao, Xin Zhao, Pengyu Gong, and Dan Feng. 2025. Revisiting network coding for warm blob storage. In *Proceedings of the 23rd USENIX Conference on File and Storage Technologies*.

[13] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. 2012. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-clouds. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*.

[14] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. 2021. Exploiting Combined Locality for Wide-Stripe Erasure Coding in Distributed Storage. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*.

[15] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, Sergey Yekhanin, et al. 2012. Erasure Coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*.

[16] Bert Hubert, Jacco Geul, and Simon Séhier. Accessed in Sept. 2025. The Wonder Shaper 1.4.1. https://github.com/magnific0/wondershaper.

[17] Saurabh Kadekodi, Shashwat Silas, David Clausen, and Arif Merchant. 2023. Practical Design Considerations for Wide Locally Recoverable Codes (LRCs). In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*.

[18] Osama Khan, Randal C Burns, James S Plank, William Pierce, and Cheng Huang. 2012. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*.

[19] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. 2018. On Fault Tolerance, Locality, and Optimality in Locally Repairable Codes. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*.

[20] Katina Kralevska, Danilo Gligoroski, Rune E Jensen, and Harald Øverby. 2018. Hashtag erasure codes: From theory to practice. *IEEE Transactions on Big Data* 4, 4 (2018), 516--529.

[21] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. 2017. Repair Pipelining for Erasure-Coded Storage. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*.

[22] Xiaolu Li, Keyun Cheng, Kaicheng Tang, Patrick PC Lee, Yuchong Hu, Dan Feng, Jie Li, and Ting-Yi Wu. 2023. ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*.

[23] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. 2019. OpenEC: Toward Unified and Configurable Erasure Coding Management in Distributed Storage Systems. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*.

[24] Xiaolu Li, Zuoru Yang, Jinhong Li, Runhui Li, Patrick P. C. Lee, Qun Huang, and Yuchong Hu. 2021. Repair pipelining for erasure-coded storage: Algorithms and evaluation. *ACM Transactions on Storage* 17, 2 (2021), 13:1--13:29.

[25] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick P. C. Lee, and Jiwu Shu. 2021. Boosting Full-Node Repair in Erasure-Coded Storage. In *Proceedings of the 2021 USENIX Annual Technical Conference*.

[26] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proceedings of the Eleventh European Conference on Computer Systems*.

[27] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, et al. 2014. f4: Facebook's Warm BLOB Storage System. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*.

[28] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Guyot, Eyal En Gad, and Zvonimir Bandic. 2016. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*.

[29] Karl Pearson. 1895. VII. Note on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London* 58, 347-352 (1895), 240--242.

[30] Andreas-Joachim Peters, Michal Kamil Simon, and Elvin Alin Sindrilaru. 2019. Erasure Coding for production in the EOS Open Storage system. In *Proceedings of 24th International Conference on Computing in High Energy and Nuclear Physics*.

[31] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies*.

[32] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. 2015. Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*.

[33] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster. In *Proceedings of the 5th USENIX conference on Hot Topics in Storage and File Systems*.

[34] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A ''Hitchhiker's'' Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proceedings of the 2014 ACM conference on SIGCOMM*.

[35] K. V. Rashmi, Nihar B. Shah, and Kannan Ramchandran. 2017. A Piggybacking Design Framework for Read-and download-efficient Distributed Storage Codes. *IEEE Transactions on Information Theory* 63, 9 (2017), 5802--5820.

[36] Irving S Reed and Gustave Solomon. 1960. Polynomial Codes over Certain Finite Fields. *J. Soc. Indust. Appl. Math.* 8, 2 (1960), 300--304.

[37] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. 2013. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the 39th International Conference on Very Large Data Bases* 6, 5 (2013), 325--336.

[38] Nihar B. Shah, K. V. Rashmi, P Vijay Kumar, and Kannan Ramchandran. 2012. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *IEEE Transactions on Information Theory* 58, 4 (2012), 2134--2158.

[39] Yingdi Shan, Kang Chen, and Yongwei Wu. 2023. Explore Data Placement Algorithm forBalanced Recovery Load Distribution.. In *Proceedings of the 2023 USENIX Annual Technical Conference*.

[40] Shan, Yingdi and Chen, Kang and Gong, Tuoyu and Zhou, Lidong and Zhou, Tai and Wu, Yongwei. 2021. Geometric Partitioning: Explore the Boundary of Optimal Erasure Code Repair. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*.

[41] Zhirong Shen, Xiaolu Li, and Patrick PC Lee. 2019. Fast predictive repair in erasure-coded storage. In *Proceedings of the 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 556--567.

[42] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. 2012. Zigzag codes: MDS Array Codes with Optimal Rebuilding. *IEEE Transactions on Information Theory* 59, 3 (2012), 1597--1616.

[43] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, et al. 2018. Clay Codes: Moulding MDS Codes to Yield an MSR Code. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*.

[44] Zhufan Wang, Guangyan Zhang, Yang Wang, Qinglin Yang, and Jiaji Zhu. 2019. Dayu: Fast and Low-interference Data Recovery in Very-large Storage Systems. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*.

[45] Hakim Weatherspoon and John D. Kubiatowicz. 2002. Erasure Coding vs. Replication: A Quantitative Comparison. In *Proceedings of International Workshop on Peer-to-Peer Systems*.

[46] Liangliang Xu, Min Lyu, Qiliang Li, Lingjiang Xie, Cheng Li, and Yinlong Xu. 2021. SelectiveEC: Towards balanced recovery load on erasure-coded storage systems. *IEEE Transactions on Parallel and Distributed Systems* 33, 10 (2021), 2386--2400.

[47] Qiaori Yao, Yuchong Hu, Xinyuan Tu, Patrick P. C. Lee Lee, Dan Feng, Xia Zhu, Xiaoyang Zhang, Zhen Yao, and Wenjia Wei. 2022. PivotRepair: Fast Pipelined Repair for Erasure-Coded Hot Storage. In *Proceedings of the42nd IEEE International Conference on Distributed Computing Systems*.

[48] Hai Zhou, Dan Feng, and Yuchong Hu. 2021. Multi-level Forwarding and Scheduling Repair Technique in Heterogeneous Network for Erasure-coded Clusters. In *Proceedings of the 50th International Conference on Parallel Processing*.

Received Dec. 30, 2024; revised May 20, 2025; accepted Sept. 13, 2025