ParaRC: Embracing Sub-Packetization for Repair Parallelization in MSR-Coded Storage

Xiaolu Li†, Keyun Cheng‡, Kaicheng Tang‡, Patrick P. C. Lee‡,
Yuchong Hu†, Dan Feng†, Jie Li*, and Ting-Yi Wu*

†*Huazhong University of Science and Technology* ‡*The Chinese University of Hong Kong*
**Huawei Technologies Co., Ltd., Hong Kong*

Abstract

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), with the minimum storage redundancy, in distributed storage systems. However, the practical repair performance of MSR codes still has significant room to improve, as the mathematical structure of MSR codes makes their repair operations difficult to parallelize. We present ParaRC, a parallel repair framework for MSR codes. ParaRC exploits the sub-packetization nature of MSR codes to parallelize the repair of sub-blocks and balance the repair load (i.e., the amount of traffic sent or received by a node) across the available nodes. We show that there exists a trade-off between the repair bandwidth and the maximum repair load, and further propose a fast heuristic that approximately minimizes the maximum repair load with limited search time for large coding parameters. We prototype our heuristic in ParaRC and show that ParaRC reduces the degraded read and full-node recovery times over the conventional centralized repair approach in MSR codes by up to 59.3% and 39.2%, respectively.

1 Introduction

Erasure coding has been widely deployed in practical distributed storage systems for providing fault tolerance against the lost data in failed storage nodes, while incurring significantly lower redundancy overhead than traditional replication [37]. Among many erasure codes, Reed-Solomon (RS) codes are the most popular and reportedly deployed in production, such as Google [11], Facebook [23], Backblaze [9], and CERN [25]. 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, thereby 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, 24, 27, 33, 36], locally repairable codes [15, 17, 32], and piggybacking codes [29, 30]. 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 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 [23, 29] (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, 24, 27, 36]); for example, Clay codes [36] 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 [3].

While MSR codes provably minimize the repair bandwidth, we argue that their practical repair performance remains bottlenecked by the node where the lost block is decoded, as the node needs to retrieve an amount of data from other available nodes more than the amount of lost data; in other words, bandwidth amplification still exists, albeit less severe than RS codes. To mitigate the repair bottleneck issue, recent studies [20, 22] 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 satisfies the additive associativity of linear combinations and the repair operation can be decomposed; in contrast, MSR 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 is to retrieve 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 distribute the repair of sub-blocks to balance the *repair load* (i.e., the amount of traffic sent or received by a node) across the available nodes.

We present ParaRC, a parallel repair framework for MSR codes that aims to balance the repair load across the available nodes and hence accelerate the repair operation. We make the following contributions:

- We observe that there exists a trade-off between the repair bandwidth and the maximum repair load. To formally analyze the trade-off, we model the repair operation of MSR codes as a directed acyclic graph (DAG) [19] and solve the repair parallelization 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 computationally expensive in general, and hence propose a fast heuristic that quickly identifies the approximate MLP point even for large coding parameters.
- We prototype ParaRC atop Hadoop 3.3.4 HDFS [4] and evaluate our prototype on Alibaba Cloud [1]. We show that ParaRC reduces the degraded read and full-node recovery times by up to 59.3% and 39.2%, respectively, compared with the centralized repair for Clay codes. We also show that ParaRC reduces the full-node recovery time of the default repair method in Hadoop-3.3.4 HDFS by 71.4%.

We release the source code of our ParaRC prototype at: **http://adslab.cse.cuhk.edu.hk/software/pararc**.

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 [4] and 256 MiB in Facebook [28]) 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.

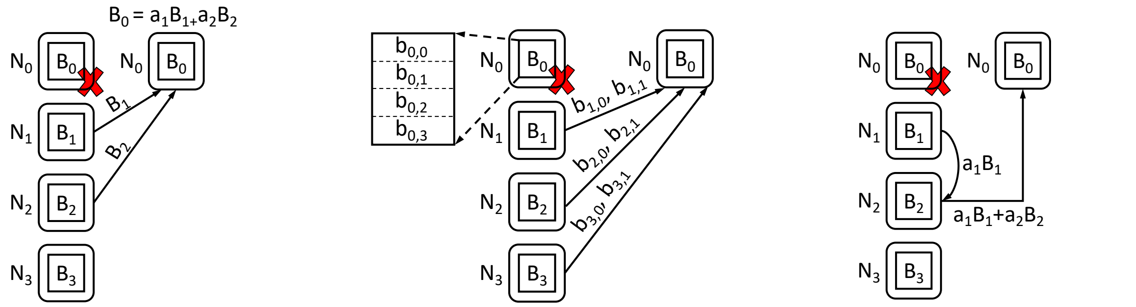
There are many approaches to construct erasure codes, among which Reed-Solomon (RS) codes [31] are the most widely deployed (e.g., [9, 11, 23, 25]). Specifically, an (n,k) RS code, configured by two parameters k and n (where $n > k$), encodes every set of k original uncoded blocks into n coded blocks, 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*, 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 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. The repair bandwidth is 512 MiB for a block size of 256 MiB.

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 512 MiB for a block size of 256 MiB.

Thus, the repair performance in RS codes is dominated by both the repair bandwidth and the 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).



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

Figure 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).

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, 24, 27, 33, 35, 36], locally repairable codes [15, 32], and piggybacking codes [29, 30]. 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 [36] and Butterfly codes [24], 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 [33])) 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 for 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 384 MiB for a block size of 256 MiB (it downloads six sub-blocks of size 64 MiB each). 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 384 MiB (same as the repair bandwidth), which is the amount of traffic downloaded in the new node.

We also consider Butterfly codes [24] 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 of size 256 MiB for the $(4,2)$ Butterfly code, the new node downloads 128 MiB of sub-blocks from each of the three available nodes, such that the repair bandwidth and the maximum repair load are both 384 MiB.

2.3 Reducing Maximum Repair Load

Some studies reduce the maximum repair load by decomposing and parallelizing a repair operation across the available nodes [20, 22]. In this work, we focus on *repair pipelining* [20], 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 [31]. 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 $(0 \leq i \leq n-1)$ 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.

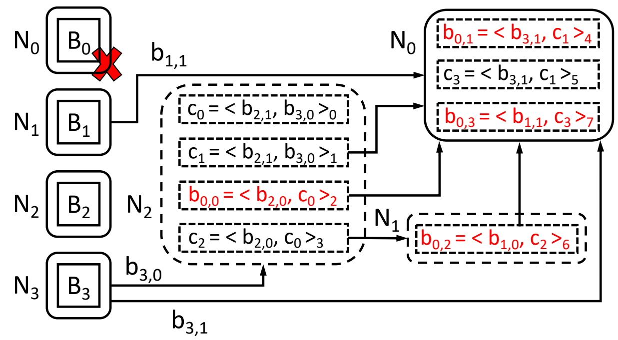
To repair B_0, repair pipelining works as follows. First, N_1 starts the repair of $b_{0,0}$ by sending $a_1 b_{1,0}$ from its local storage to N_2. Second, N_2 combines the received $a_1 b_{1,0}$ with $a_2 b_{2,0}$ from its local storage to form $a_1 b_{1,0} + a_2 b_{2,0}$. Third, N_2 sends $a_1 b_{1,0} + a_2 b_{2,0}$ 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 256 MiB for a block size of 256 MiB 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 512 MiB, the same as in the conventional repair for RS codes (Figure 1(a)), since k available nodes transfer k blocks of data in total.

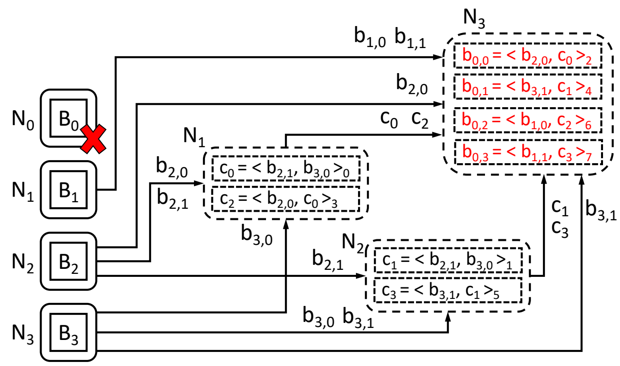
2.4 Motivation and Challenges

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).



(a) Repair bandwidth = 448 MiB; Max. repair load = 320 MiB



(b) Repair bandwidth = 832 MiB; Max. repair load = 512 MiB

Figure 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).

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

	Repair bandwidth (MiB)	Maximum repair load (MiB)
RS; centralized	512 (highest)	512 (highest)
Clay; centralized	384 (lowest)	384 (high)
RS; parallel	512 (highest)	256 (lowest)
Clay; parallel	448 (medium)	320 (medium)

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

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). 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 320 MiB (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) for a block size of 256 MiB.

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 384 MiB to 448 MiB.

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 832 MiB, while the maximum repair load is 512 MiB.

In summary, the parallel repair of MSR codes can be scheduled to balance the trade-off between the repair bandwidth and the 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 solution that can effectively balance the trade-off for general coding parameters of MSR codes.

3 Model and Analysis

Before we design the parallel repair solution for MSR codes, we first formulate a generic repair model that characterizes the trade-offs between the repair bandwidth and the 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 the repair bandwidth and the maximum repair load of a repair solution (§3.2). Finally, we analyze the trade-off between the repair bandwidth and the maximum repair load for different repair solutions on RS and MSR codes (§3.3).

3.1 Characterizing Repair Solutions

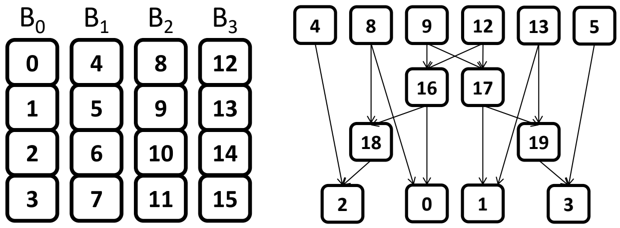
Design requirements. We first identify three design requirements for our repair model to characterize repair solutions based on our example in Figure 2:

- 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).

Our repair model builds on the ECDAG abstraction [19], 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}. 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 [36] 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



(a) Layout (b) ECDAG

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

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.

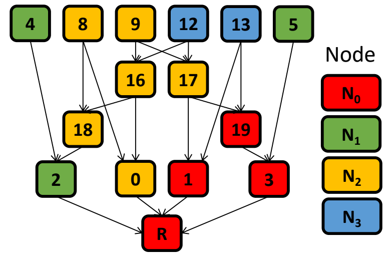
pECDAG. We extend the ECDAG abstraction into the pECDAG abstraction to support the scheduling of parallel sub-block repair operations, so that we can model the trade-off between the repair bandwidth and the maximum repair load. Specifically, a pECDAG makes two extensions over an ECDAG. First, it associates 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. To help our discussion, we refer to the topmost vertices (e.g., v_4, v_5, v_8, v_9, v_{12}, and v_{13}) that correspond to the sub-blocks retrieved from the other available nodes as the *leaf vertices*, and the vertex R that corresponds to the data collector as the *root vertex*. Note that the colors of both the leaf vertices and root vertex are fixed, as they depend on where the retrieved sub-blocks and repaired block reside, respectively.

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 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.

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. We associate each pECDAG instance with a *traffic table*, so as to efficiently quantify the repair bandwidth and the 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



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

(a) pECDAG (b) Load table

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

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.

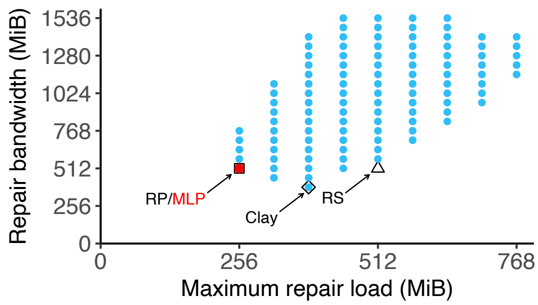
Construction of a traffic table. We show how we generate the traffic table for a given pECDAG instance. We initialize a traffic table T with two arrays $T.In$ and $T.Out$, which record the numbers of incoming and outgoing sub-blocks for each node, respectively. For each vertex v_i, we traverse each edge $e(v_i,v_j)$. 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 the traffic table. As v_{19} and v_1 have the same color, we do not need to increment the traffic table for $e(v_{17},v_{19})$.

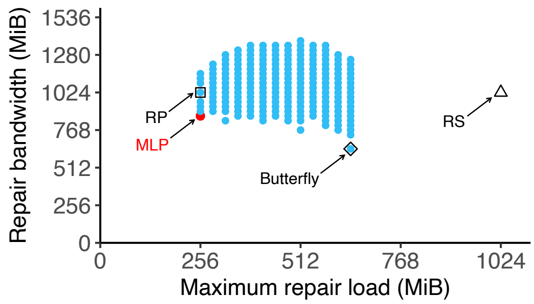
3.3 Trade-off Analysis

Based on a pECDAG and its traffic table, we study the trade-off between the repair bandwidth and the maximum repair load. Our idea is to enumerate all possible color combinations of a pECDAG 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



(a) $(4,2)$ Clay code ($w=4$)



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

Figure 5: Trade-off analysis between the repair bandwidth and the maximum repair load.

$(4,2)$ 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 have different trade-offs between the repair bandwidth and the maximum repair load. We define a *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 value of repair bandwidth. For example, for the $(4,2)$ Clay code, the MLP happens to be overlapped with the point of RP; for the $(6,4)$ Butterfly code, the MLP reduces the repair bandwidth by 15.6% compared with that of RP, while it achieves 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 size of the search space, and hence the running time.

4 Heuristic

As the brute-force approach in general is time-consuming to find the MLP, we propose a heuristic to find an approximate point that is close to the MLP. Our goal is to find a parallel repair solution represented in a pECDAG that keeps both the repair bandwidth and the maximum repair load as low as possible.

Design idea. The high-level idea is to search all the color combinations for a pECDAG, while pruning some branches based on the heuristic to reduce the search space. Intuitively, we can view our heuristic as searching for the solution based on Pareto optimality, such that it searches for the MLP on the Pareto frontier and prunes the dominated solutions that have both larger repair bandwidth and larger maximum repair load than a candidate solution.

We first introduce the key definitions. If two pECDAGs, say X and Y, have the same DAG structure except in the color of a single vertex that refers to an intermediate sub-block or a repaired sub-block, we call X and Y the *neighbors*. We perform the search on a pECDAG by examining all the neighbors of the pECDAG. If we have examined all the neighbors of a pECDAG, we say that the pECDAG is *searched*; otherwise, the pECDAG is *un-searched*. Our heuristic is composed of the following three steps.

Step 1: Initialization. We define an *un-searched pool*, which is used to keep the pECDAGs that will be searched, as well as a *candidate pool*, which is used to record the candidate pECDAG solutions to be returned. At the beginning, we generate a random pECDAG, in which the color of a vertex that refers to an intermediate sub-block or a repaired sub-block is randomly selected from a set of candidate colors that represent the nodes storing the available blocks and the node storing the repaired block. We add the random pECDAG to the un-searched pool and the candidate pool for initialization.

Step 2: Searching. Each time we retrieve a pECDAG from the un-searched pool. We enumerate all the neighbors of this pECDAG by changing the color of only one vertex (which refers to an intermediate sub-block or a repaired sub-block). If there are α such vertices and β candidate colors, a pECDAG has $\alpha \times (\beta - 1)$ neighbors (note that for each vertex, there are $\beta - 1$ different candidate colors to which we can change). After we examine all the neighbors of the pECDAG (i.e., the pECDAG is searched), we remove the pECDAG from the un-searched pool.

Step 3: Pruning. After Step 2, we generate $\alpha \times (\beta - 1)$ new neighbors of a pECDAG. However, not all of them are suitable for future search. Here, we consider different cases of how we compare a neighbor pECDAG that we generate in Step 2 with the pECDAGs in the candidate pool to decide whether the neighbor pECDAG is suitable for future search. Suppose that there are two pECDAGs in the candidate pool (say, A and B), and Figure 6 shows the four cases when we compare a neighbor pECDAG with the solutions in the candidate pool:

- Case 1 (Figure 6(a)): This is an example of a generated neighbor pECDAG that is not suitable for future search. If there exists a pECDAG in the candidate pool that provides less maximum repair load and less repair bandwidth than the neighbor that we generate in Step 2, it means that we already have an existing solution that outperforms the neighbor. Thus, we discard the neighbor. The remaining three cases show the examples of when we can add a neighbor pECDAG to the candidate pool.
- Case 2 (Figure 6(b)): If the neighbor has the least maximum repair load or the least repair bandwidth compared with all the pECDAGs in the candidate pool, we can add the neighbor to the candidate pool.
- Case 3 (Figure 6(c)): If the neighbor lies between two solutions in the candidate pool, we can add the neighbor to the candidate pool.
- Case 4 (Figure 6(d)): If we find that the repair bandwidth and the maximum repair load of the neighbor are both less than those of an existing pECDAG in the candidate pool, we can add the neighbor to the candidate pool and also remove the existing one from the candidate pool.

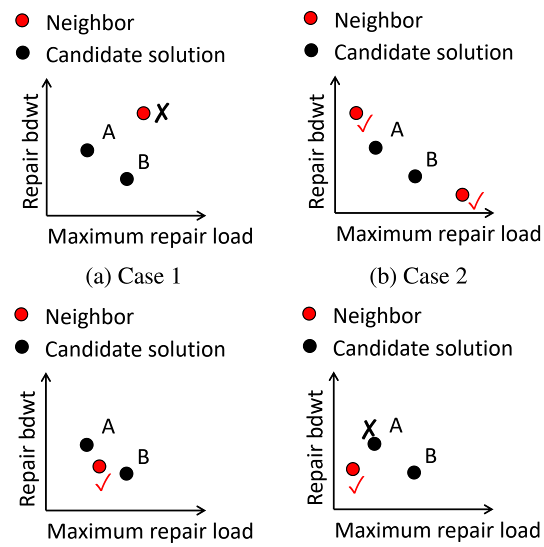
For the neighbors that have been added to the candidate pool, we also add them to the un-searched pool for our future search.

Note that Step 2 and Step 3 are performed iteratively until the un-searched pool is empty. Then, we report the pECDAG that has the least maximum repair load as an approximate MLP from the candidate pool.

Discussion. As our heuristic in general only provides a local optimal solution, we can repeat the process to find an approximate MLP starting from Step 1 by multiple runs, such that we can choose the one with the least maximum repair load from all the runs. We show in §7.2 how the heuristic performs in finding the approximate MLP.

5 Design of Repair Operations

Repair operations in a distributed storage system will be triggered in two scenarios, namely *degraded reads*, where a client reads an unavailable block, and *full-node recovery*, where the distributed storage system repairs the lost blocks of a failed node in a new node. In this section, we design the two repair operations based on our heuristic in §4.



(a) Case 1 (b) Case 2 (c) Case 3 (d) Case 4

Figure 6: Compare a neighbor generated in Step 2 with solutions in the candidate pool.

Degrade reads. A client issues a degraded read operation when it requests an unavailable block, in which it needs to repair the requested block through the available blocks of the same stripe stored in other nodes. We generate an approximate MLP from our heuristic. We then associate the approximate MLP with a pECDAG, which describes the repair workflow with the sub-blocks (including the available sub-blocks, repaired sub-blocks, and intermediate sub-blocks) and the nodes (i.e., the nodes that store the available blocks and the node associated with client). Note that a pECDAG varies for different unavailable blocks of a stripe. Also, as the blocks of different stripes are distributed across different sets of nodes in a distributed storage system, a pECDAG varies across different stripes and needs to be generated for each requested unavailable block of a stripe.

Full-node recovery. In a full-node recovery operation, a new node is added to the system, and we repair all the lost blocks of a failed node and store the repaired blocks in the new node. We run our heuristic to generate an approximate MLP for each lost block to be repaired and associate the approximate MLP with a pECDAG. For each block, we associate the colors in the corresponding pECDAG with both the nodes that store the available blocks in the stripe and the new node that is added for full-node recovery.

6 ParaRC

We propose a parallel repair middleware, ParaRC, to balance the repair bandwidth and the maximum repair load for MSR codes. We first introduce the architecture of ParaRC in §6.1. We then elaborate on the implementation details in §6.2.

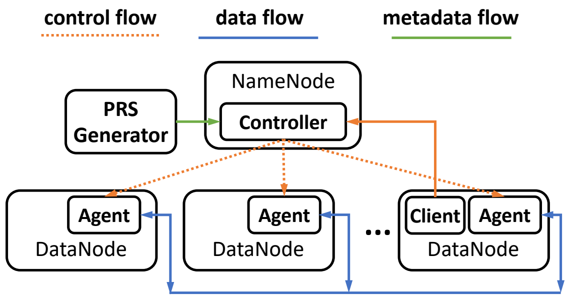


Figure 7: System architecture.

6.1 Architecture

We have built ParaRC as a repair middleware based on OpenEC [19], an erasure coding framework that supports the deployment of custom erasure coding solutions in existing distributed storage systems. ParaRC leverages OpenEC to deploy the parallel repair of MSR codes on Hadoop HDFS [4]. HDFS stores data in fixed-size blocks. It 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 the storage for the blocks. ParaRC performs encoding across HDFS blocks: for an (n,k) code, it encodes every k uncoded HDFS blocks (i.e., data blocks) into $n-k$ coded HDFS blocks (i.e., parity blocks) to form a stripe, and stores the stripe in n DataNodes.

Figure 7 shows the architecture of ParaRC when it is integrated with HDFS. ParaRC includes a parallel repair solution generator, called the *PRS generator*. It also deploys a *controller* that runs within the NameNode, and multiple *agents*, each of which runs within a DataNode. We also deploy a *client* that is co-located with an agent in a DataNode to issue repair requests to ParaRC (note that the client can also run in a standalone machine outside of the DataNodes). We now elaborate on each component in detail.

PRS generator. The PRS generator pre-computes the parallel repair solution for each single-block repair scenario *offline* and stores the results before the system starts [16]; 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 [26]. The PRS generator runs the heuristic in §4 for an MSR code to generate a parallel repair solution that operates at an approximate MLP. It constructs a pECDAG based on the parallel repair solution. It stores the solution in the controller, which coordinates the actual repair operation.

Controller. The controller coordinates the parallel repair operation for the lost blocks that are encoded with MSR codes. Upon receiving a repair request for a block, the controller first reads the metadata of the block from HDFS to determine the location of other blocks in the same stripe. Then, the controller constructs a pECDAG to repair the block with the parallel repair solution returned from the PRS generator. Finally, it translates the pECDAG into a set of *basic tasks* defined in OpenEC [19], including (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 block. Then, the controller sends the basic tasks to the agents to perform sub-block repair operations to repair a lost block.

Agent. Each agent performs the basic tasks assigned by the controller. For a reading task, an agent directly reads the sub-blocks of a block stored in the local file system. For a fetching task, an agent downloads the sub-blocks from another agent. For a computing task, the agent generates the intermediate sub-blocks or repaired sub-blocks. For a persisting task, an agent stores the repaired sub-blocks as the final repaired block.

Client. A client sends repair requests to ParaRC. It can send a degraded read request or a full-node recovery request to ParaRC (§5). For a degraded read request, ParaRC coordinates the parallel repair for an unavailable block requested by the client; for a full-node recovery request, ParaRC repairs all lost blocks of a failed DataNode in parallel in a new DataNode.

6.2 Implementation

We implement ParaRC in C++ with around 9.4 K LoC and integrate ParaRC with Hadoop-3.3.4 HDFS [4] (HDFS-3 for short). ParaRC uses Redis [7] for internal communications among the controller, agents, and clients. It uses Intel's Intelligent Storage Acceleration Library (ISA-L) [6] to perform encoding and decoding operations for erasure codes. It supports both the centralized repair and parallel repair for MSR codes. In the following, we elaborate on the deployment details of ParaRC and how ParaRC is integrated with HDFS-3.

Deployment. 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 we deploy the pECDAG in Figure 4(a), we need to co-locate the repair of sub-blocks v_{18} and v_0, to make sure that the sub-blocks v_8 and v_{16} are only downloaded once in N_2 in the sub-block repair operation. To achieve this goal, we first divide 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 in 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, such that N_2 can only download sub-block v_{12} from N_3 only 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, in ParaRC, the encoding of a stripe of blocks is divided into the encoding of multiple small sub-stripes, where the data unit in each node within 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 [19], so that ParaRC can encode different sub-stripes in parallel to fully utilize the system resources.

Note that HDFS-3 does not directly support MSR codes, so we rely on ParaRC to generate MSR-coded blocks and store them in HDFS-3. To enable the parallel repair for MSR codes in HDFS-3, we run the ParaRC controller with the NameNode and run each ParaRC agent with 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 in the same stripe. We store the metadata of HDFS-3 blocks in the stripe store of ParaRC, such that when repairing a block, the controller can retrieve metadata from the stripe store.

Support for RS codes. ParaRC also supports RS codes. It now implements both the conventional centralized repair approach and the parallel repair approach based on repair pipelining [20]. 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 [20]). The corresponding parallel repair solutions are stored in the PRS generator. Note that RS codes have no sub-packetization and a sub-stripe encodes k packets into n RS-coded packets.

7 Evaluation

We conduct experiments for ParaRC on Alibaba Cloud [1]. We aim to answer the following questions:

- What is the performance of our heuristic in §4 in finding the approximate MLP? (§7.2)
- How does the performance of ParaRC vary across different system configurations? (§7.3 and §7.4)
- What is the performance overhead of ParaRC to HDFS-3 and how is the repair performance improved by the parallel repair from ParaRC? (§7.5)

7.1 Setup

Testbed. We provision 23 memory-optimized instances on Alibaba Cloud [1] for ParaRC, which includes the PRS generator, the controller, 20 agents, and a node that serves as the client for degraded reads or the new node for full-node recovery. The PRS generator runs on an `ecs.r7.2xlarge` instance with 8 vCPUs and 64 GiB RAM, while other components are deployed in `ecs.r7.xlarge` instances with 4 vCPUs and 16 GiB RAM. Each instance is also equipped with a 40 GiB enhanced SSD with performance level PL0 [2] and is installed with Ubuntu 18.04. All instances are connected via a 10 Gbps network.

Default settings. We configure the default block size as 256 MiB and the default sub-packet size as 64 KiB; for example, the packet size for the $(14,10)$ Clay code is 256×64 KiB $= 16$ MiB, so a stripe can be divided into 16 sub-stripes. In our evaluation, we compare four repair approaches: (i) the centralized repair for RS codes (RS); (ii) repair pipelining for RS codes (RP); (iii) the centralized repair for Clay/Butterfly codes (Clay/Butterfly); and (iv) the parallel repair for Clay/Butterfly codes (ParaRC). If we consider an (n,k) Clay/Butterfly code, we also use the same (n,k) for RS and RP.

For degraded reads, we evaluate the average degraded read time for the first k uncoded blocks. For full-node recovery, 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). We plot the average results over 5 runs, including the error bars showing the maximum and minimum of the 5 runs.

7.2 Finding the Approximate MLP

E1: Performance of finding the approximate MLP. We evaluate our heuristic in §4 in finding the approximate MLP. We focus on repairing B_0 for Clay codes [36] and Butterfly codes [24]. We evaluate the algorithm running times of our heuristic and the brute-force approach. We also compare the maximum repair load and repair bandwidth of the approximate MLP returned by our heuristic with those of RP and the centralized repair for Clay/Butterfly codes.

We first compare the running time of our heuristic with that of the brute-force approach. We only consider the $(4,2)$ Clay code ($w = 4$) and the $(6,4)$ Butterfly code ($w = 8$), as the brute-force approach for large (n,k) cannot be solved within reasonable time. As shown in Table 2, for the $(4,2)$ Clay code, the heuristic reduces the running time from 264.1 s to 1.8 s, while for the $(6,4)$ Butterfly code, the heuristic reduces the running time from 34.2 s to 0.3 s. We also examine the number of pECDAGs being examined by the heuristic. For example, for the $(14,10)$ Clay code, the heuristic examines about 14 million pECDAGs only; the number is much less than the lower bound of the number of pECDAGs (i.e., 14^{256}) that need to be examined by the brute-force approach (§3.3). Thus, the heuristic significantly reduces the search space.

We note that the heuristic can find the solution whose maximum repair load has the same size as the block size, but sometimes cannot. For example, the solution for the $(6,4)$ Butterfly code ($w = 8$) achieves the maximum repair load of 256 MiB (which is also the minimum), while the maximum repair load of the solution for the $(4,2)$ Clay code ($w = 4$) is larger than the block size 256 MiB. The reason is that the heuristic may return a local optimal solution.

We then compare the maximum repair load and repair bandwidth of different repair approaches. The maximum repair load of our heuristic is significantly less than that of the centralized repair for Clay/Butterfly codes. For example, for the $(14,10)$ Clay code ($w = 256$) the maximum repair load of the MLP decreases to 271 MiB, which is 67.4% less than that of the centralized repair for Clay codes (i.e., 832 MiB). We also

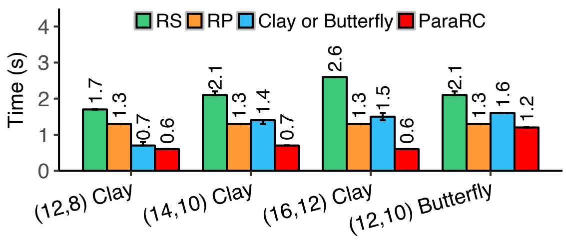
(n,k,w)	**RP**	**Clay**	**Approximate MLP**	**Heuristic**	**Brute-force**
(4,2,4)	(256,512)	(384,384)	(320,448)	1.8 s	264.1 s
(12,8,64)	(256,2048)	(704,704)	(264,1224)	425.9 s	-
(14,10,256)	(256,2560)	(832,832)	(271,1609)	57.2 h	-
(16,12,256)	(256,3072)	(960,960)	(281,1774)	61.9 h	-

(a) Clay codes

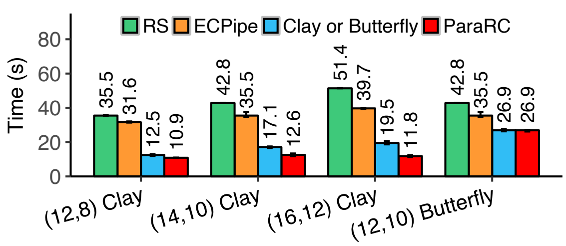
(n,k,w)	**RP**	**Butterfly**	**Approximate MLP**	**Heuristic**	**Brute-force**
(6,4,8)	(256,1024)	(640,640)	(256,896)	0.3 s	34.2 s
(12,10,512)	(256,2560)	(1408,1408)	(297,2216)	31.64 h	-

(b) Butterfly codes

Table 2: E1: Performance of finding the approximate MLP to repair B_0 for Clay codes and Butterfly codes. We show the (maximum repair load, repair bandwidth) for each repair approach. We also show the running time for the heuristic. For the brute-force approach, we only show the running time for the $(4,2,4)$ Clay code and $(6,4,8)$ Butterfly code, as the other configurations cannot be completed within reasonable time.



(a) Degraded reads



(b) Full-node recovery

Figure 8: E2: Varying MSR codes.

observe that when the maximum repair load decreases, the repair bandwidth of our heuristic is higher than that of the centralized repair, while it is still much less than the repair bandwidth of RP (by 37.1%). We also observe similar trends in Butterfly codes.

7.3 ParaRC Performance

We evaluate the performance of ParaRC in degraded reads and full-node recovery under different settings.

E2: Varying MSR codes. We evaluate the performance of ParaRC 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 8 shows the evaluation results.

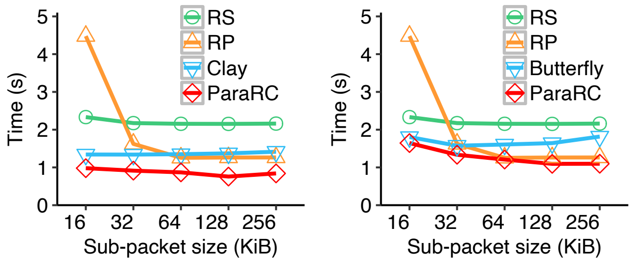
We first analyze the performance of the degraded reads, as shown in Figure 8(a). Overall, ParaRC has the smallest degraded read time compared with other baseline repair approaches. For example, for the $(16,12)$ Clay code, ParaRC reduces the degraded read time by 76.4%, 51.9%, and 59.3%, compared with RS, RP, and Clay, respectively. Although RP minimizes the maximum repair load, its degraded read time is not necessarily minimized, as RP still has high repair bandwidth and needs to read the whole block from each available node in degraded reads. For the $(12,10)$ Butterfly code, ParaRC reduces the degraded read time by 43.8%, 3.7%, and 24.8% compared with RS, RP, and Butterfly, respectively.

We next analyze the performance of full-node recovery, as shown in Figure 8(b). Like degraded reads, ParaRC also has the smallest full-node recovery time compared with other baseline repair approaches. For example, for the $(16,12)$ Clay code, ParaRC reduces the full-node recovery time by 76.9%, 70.2%, and 39.2% compared with RS, RP, and Clay, respectively. For the $(12,10)$ Butterfly code, ParaRC reduces the full-node recovery time by 37.2% and 24.2% compared with RS and RP, respectively. We observe that the network bandwidth usages of the centralized repair for the $(12,8)$ Clay code, the $(14,10)$ Clay code, the $(16,12)$ Clay code, and the $(12,10)$ Butterfly code are 1,126 MiB/s, 973 MiB/s, 984 MiB/s, and 1,046 MiB/s, respectively, implying that it is bottlenecked by the high maximum repair load at the new node where the lost blocks are reconstructed. As ParaRC reduces the maximum repair load, we observe that it significantly improves the repair performance for Clay codes. However, we also observe that the performance improvement of ParaRC is marginal for Butterfly codes. The reason is that while the maximum repair load reduces for Butterfly codes, the repair bandwidth also increases (i.e., from 1,408 MiB to 2,216 MiB), thereby limiting the performance improvements.

7.4 Micro-benchmarks

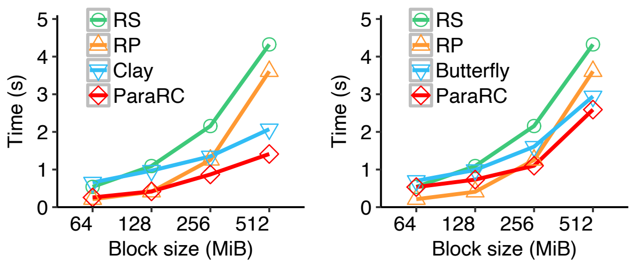
We study how the performance of ParaRC varies for different sub-packet sizes and block sizes. We focus on the $(14,10)$ Clay code and the $(12,10)$ Butterfly code.

E3: Varying sub-packet size for degraded reads. We evaluate ParaRC 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).



(a) (14,10) Clay code (b) (12,10) Butterfly code

Figure 9: E3: Varying sub-packet size for degraded reads.



(a) (14,10) Clay code (b) (12,10) Butterfly code

Figure 10: E4: Varying block size for degraded reads.

Figure 9 shows the results. For the (14,10) Clay code, ParaRC has the smallest degraded read time compared with other repair approaches for all the sub-packet sizes being considered. For example, when the sub-packet size is 128 KiB, ParaRC reduces the degraded read time by 64.8%, 40.0%, and 44.8% compared with RS, RP, and Clay, respectively. For each repair approach, we see a performance drop when the sub-packet size decreases to 16 KiB due to the overhead of processing a large number of sub-packets, and such a performance drop is especially significant for RP. For example, when we decrease the sub-packet size from 64 KiB to 16 KiB, the degraded read time of ParaRC increases by 12.8%. However, ParaRC still outperforms other baseline repair approaches.

For the Butterfly code, we also observe similar results. For all sub-packet sizes, ParaRC has the smallest degraded read time. For example, when the sub-packet size is 128 KiB, ParaRC reduces the degraded read time by 49.2%, 13.3%, and 33.4% compared with RS, RP, and Butterfly, respectively. When the sub-packet size decreases from 64 KiB to 16 KiB, the degraded read time increases by 26.5%.

E4: Varying block size. We evaluate ParaRC 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 (§7.1).

Figure 10 shows the results. When the block size is large, ParaRC outperforms other repair approaches. For example, when the block size is 512 MiB, ParaRC for the (14,10) Clay code reduces the degraded read time by 67.3%, 60.9%, and 31.8% compared with RS, RP, and Clay, respectively, while

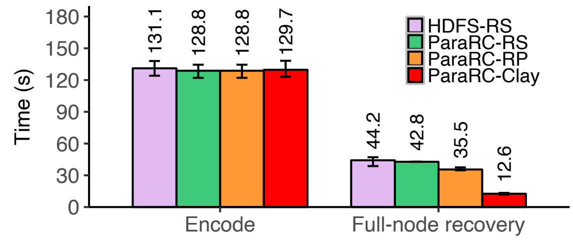


Figure 11: E5: HDFS-3 integration.

ParaRC for the (12,10) Butterfly code reduces the degraded read time by 40.3%, 28.4%, and 12.3% compared with RS, RP, and Butterfly, respectively.

When the block size is small, RP outperforms ParaRC. For example, when the block size is 64 MiB, ParaRC for the (14,10) Clay code has 22.1% higher degraded read time than RP. The reason is that ParaRC fails to benefit from the parallel repair for small block sizes due to high sub-packetization. For example, when the block size is 64 MiB, a stripe is only divided into 4 sub-stripes that are repaired in parallel for the (14,10) Clay code, so the degree of parallelism is low. In contrast, RP can pipeline the repair of 1,024 sub-stripes in parallel (§6.2) and outperform ParaRC for small block sizes.

7.5 Performance in HDFS-3

E5: HDFS-3 integration. We evaluate the integration of ParaRC into HDFS-3. Recall that we have shown the benefits of ParaRC over other repair approaches (E2-E4). In this experiment, we only focus on the performance overhead and performance gain of ParaRC 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 ParaRC with the RS code implementation in HDFS-3. We focus on evaluating the overhead of encoding data by Clay codes in ParaRC and the full-node recovery performance gain of ParaRC. We omit the results for degraded reads to a lost block. The reason is that in HDFS-3, a degraded read is triggered when reading a file, where all blocks of the original file are returned to the client anyway. In this case, the centralized repair of the lost block is sufficient.

We evaluate the performance of encoding 20 stripes and repairing 20 lost blocks of a failed node in full-node recovery (the full-node recovery procedure is described in §7.1). 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 ParaRC (denoted by ParaRC-RS); (iii) encoding by RS codes and performing repair pipelining [20] in ParaRC (denoted by ParaRC-RP); and (iv) encoding by Clay codes and performing the parallel repair in ParaRC (denoted by ParaRC-Clay).

Figure 11 shows the results. For encoding, we observe that the encoding of Clay codes in ParaRC and that of the encoding of RS codes in HDFS-3 have similar overhead. For example, HDFS-RS takes 131.1 s, while ParaRC-Clay takes 129.7 s for encoding 20 stripes. For full-node recovery, ParaRC-Clay reduces the full-node recovery time by 71.4% compared with HDFS-RS; note that the total repair bandwidth for the full-node recovery of 20 lost blocks in ParaRC-Clay is 16.25 GiB, while that in HDFS-RS is 50 GiB (where $(n,k) = (14,10)$).

8 Related Work

RS codes [31] are popularly deployed in distributed storage systems [3, 5, 9, 11, 23, 25], 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 [22] divides the repair of a block into partial operations and parallelizes them for improved repair performance. Repair pipelining [18, 20] 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 [8], SMFRepair [39], and PivotRepair [38] 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 [27], Butterfly codes [24], and Clay codes [36]. Such MSR codes operate in different parameter regimes, such as $n-k=2$ for F-MSR codes [13] and Butterfly codes [24], and $n \geq 2k-1$ for PM-RBT codes [27]. In particular, Clay codes [36] 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 [34] 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 [19] 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 [21] proposes a DAG abstraction to load-balance the full-node recovery workflow. Our work extends ECDAG [19] to support the parallel repair for MSR codes.

9 Conclusions and Future Work

We present ParaRC, a parallel repair framework that improves the repair performance of MSR-coded distributed storage systems by exploiting the sub-packetization nature of MSR codes. We show that there is a trade-off between the repair bandwidth and the maximum repair load. ParaRC builds on a fast heuristic that aims to minimize the maximum repair load, while maintaining the low repair bandwidth. We implement ParaRC as a middleware that runs atop HDFS. Our evaluation results on Alibaba Cloud demonstrate that ParaRC improves the repair performance of degraded reads and full-node recovery compared with the conventional centralized repair approach for Clay codes and Butterfly codes as well as the repair pipelining approach for RS codes.

We discuss the limitations of our work and pose the following open issues for future work.

- Currently, we only empirically show the performance gain of ParaRC, but the theoretical analysis of the ParaRC design remains unexplored. Some open theoretical issues include: (i) the formulation of a multi-objective optimization problem that minimizes both the repair bandwidth and the maximum repair load; (ii) the difference between the returned solution of the heuristic in §4 and the MLP; (iii) the convergence of the heuristic in §4 to the MLP; and (iv) faster and more efficient heuristics.
- ParaRC now focuses on the repair parallelism within a single stripe (i.e., intra-stripe parallelism). One optimization is to extend ParaRC with the repair parallelism across multiple stripes (i.e., inter-stripe parallelism) for further performance gains, particularly in declustered settings where the stripes span across a large number of nodes [12]. Also, the full-node recovery of ParaRC currently stores all reconstructed blocks in a new node that replaces the failed node. We can extend it to distribute the reconstructed blocks across different nodes to avoid bottlenecking the new node.
- ParaRC is currently designed for large blocks and the moderate parameters n and k. In future work, we consider small blocks and wide stripes [14] (wide stripes encode data with large parameters n and k for ultra-low redundancy).

Acknowledgments. We thank our shepherd, Gala Yadgar, and the anonymous reviewers for their comments. This work was supported in part by the National Key Research and Development Program of China for Young Scholars (No. 2021YFB0301400), Key Laboratory of Information Storage System Ministry of Education of China, the National Natural Science Foundation of China (No. 61821003 and No. 61832007), and Research Grants Council of HKSAR (AoE/P-404/18). The corresponding author is Yuchong Hu.

References

[1] Alibaba Cloud. https://www.alibabacloud.com/product/ecs-pricing-list/en.

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

[3] Ceph - Erasure code. http://docs.ceph.com/docs/master/rados/operations/erasure-code/.

[4] Hadoop 3.3.4. https://hadoop.apache.org/docs/r3.3.4/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[5] HDFS erasure coding. https://hadoop.apache.org/docs/r3.0.0/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html.

[6] Intel's intelligent storage acceleration library (ISA-L). https://www.intel.com/content/www/us/en/developer/tools/isa-l/overview.html.

[7] redis.io. https://redis.io/.

[8] Yunren Bai, Zihan Xu, Haixia Wang, and Dongsheng Wang. Fast recovery techniques for erasure-coded clusters in non-uniform traffic network. In *Proc. of ICPP*, 2019.

[9] Brian Beach. Backblaze Vaults: Zettabyte-scale cloud storage architecture. https://www.backblaze.com/blog/vault-cloud-storage-architecture/, 2019.

[10] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.

[11] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *Proc. of USENIX OSDI*, 2010.

[12] Mark Holland, Garth A. Gibson, and Daniel P. Siewiorek. Architectures and algorithms for on-line failure recovery in redundant disk arrays. *Distributed Parallel Databases*, 2(3):295–335, 1994.

[13] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proc. of USENIX FAST*, 2012.

[14] Yuchong Hu, Liangfeng Cheng, Qiaori Yao, Patrick P. C. Lee, Weichun Wang, and Wei Chen. Exploiting combined locality for wide-stripe erasure coding in distributed storage. In *Proc. of USENIX FAST*, 2021.

[15] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[16] Osama Khan, Randal C Burns, James S. Plank, William Pierce, and Cheng Huang. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

[17] Oleg Kolosov, Gala Yadgar, Matan Liram, Itzhak Tamo, and Alexander Barg. On fault tolerance, locality, and optimality in locally repairable codes. In *Proc. of USENIX ATC*, 2018.

[18] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[19] Xiaolu Li, Runhui Li, Patrick P. C. Lee, and Yuchong Hu. OpenEC: Toward unified and configurable erasure coding management in distributed storage systems. In *Proc. of USENIX FAST*, 2019.

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

[21] Shiyao Lin, Guowen Gong, Zhirong Shen, Patrick P. C. Lee, and Jiwu Shu. Boosting full-node repair in erasure-coded storage. In *Proc. of USENIX ATC*, 2021.

[22] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proc. of ACM EuroSys*, 2016.

[23] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm blob storage system. In *Proc. of USENIX OSDI*, 2014.

[24] Lluis Pamies-Juarez, Filip Blagojevic, Robert Mateescu, Cyril Guyot, Eyal En Gad, and Zvonimir Bandic. Opening the chrysalis: On the real repair performance of msr codes. In *Proc. of USENIX FAST*, 2016.

[25] Andreas-Joachim Peters, Michal Kamil Simon, and Elvin Alin Sindrilaru. Erasure coding for production in the EOS open storage system. In *Proc. of CHEP*, 2019.

[26] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. A performance evaluation and examination of open-source erasure coding libraries for storage. In *Proc. of USENIX FAST*, 2009.

[27] K. V. Rashmi, Preetum Nakkiran, Jingyan Wang, Nihar B. Shah, and Kannan Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[28] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the Facebook warehouse cluster. In *Proc. of USENIX HotStorage*, 2013.

[29] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. of ACM SIGCOMM*, 2014.

[30] K. V. Rashmi, Nihar B. Shah, and Kannan Ramchandran. A piggybacking design framework for read-and download-efficient distributed storage codes. *IEEE Trans. on Information Theory*, 63(9):5802–5820, 2017.

[31] Irving S. Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[32] Maheswaran Sathiamoorthy, Megasthenis Asteris, Dimitris Papailiopoulos, Alexandros G Dimakis, Ramkumar Vadali, Scott Chen, and Dhruba Borthakur. XORing elephants: Novel erasure codes for big data. *Proc. of the VLDB Endowment*, 6(5):325–336, 2013.

[33] Nihar B. Shah, K. V. Rashmi, P. Vijay Kumar, and Kannan Ramchandran. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Trans. on Information Theory*, 58(4):2134–2158, 2012.

[34] Yingdi Shan, Kang Chen, Tuoyu Gong, Lidong Zhou, Tai Zhou, and Yongwei Wu. Geometric partitioning: Explore the boundary of optimal erasure code repair. In *Proc. of ACM SOSP*, 2021.

[35] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Trans. on Information Theory*, 2012.

[36] Myna Vajha, Vinayak Ramkumar, Bhagyashree Puranik, Ganesh Kini, Elita Lobo, Birenjith Sasidharan, P. Vijay Kumar, Alexandar Barg, Min Ye, Srinivasan Narayanamurthy, Syed Hussain, and Siddhartha Nandi. Clay codes: Moulding MDS codes to yield an MSR code. In *Proc. of USENIX FAST*, 2018.

[37] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. of IPTPS*, 2002.

[38] Qiaori Yao, Yuchong Hu, Xinyuan Tu, Patrick P. C. Lee Lee, Dan Feng, Xia Zhu, Xiaoyang Zhang, Zhen Yao, and Wenjia Wei. PivotRepair: Fast pipelined repair for erasure-coded hot storage. In *Proc. of ICDCS*, 2022.

[39] Hai Zhou, Dan Feng, and Yuchong Hu. Multi-level forwarding and scheduling repair technique in heterogeneous network for erasure-coded clusters. In *Proc. of ICPP*, 2021.