Boosting Full-Node Repair in Erasure-Coded Storage

Shiyao Lin[1], Guowen Gong[1], Zhirong Shen[1]*, Patrick P. C. Lee[2], and Jiwu Shu[1,3]
[1]*Xiamen University* [2]*The Chinese University of Hong Kong* [3]*Tsinghua University*

Abstract

As a common choice for fault tolerance in today's storage systems, erasure coding is still hampered by the induced substantial traffic in repair. A variety of erasure codes and repair algorithms are designed in recent years to relieve the repair traffic, yet we unveil via careful analysis that they are still plagued by several limitations, which restrict or even negate the performance gains. We present RepairBoost, a scheduling framework that can assist existing linear erasure codes and repair algorithms to boost the full-node repair performance. RepairBoost builds on three design primitives: (i) repair abstraction, which employs a directed acyclic graph to characterize a single-chunk repair process; (ii) repair traffic balancing, which balances the upload and download repair traffic simultaneously; and (iii) transmission scheduling, which carefully dispatches the requested chunks to saturate the most unoccupied bandwidth. Extensive experiments on Amazon EC2 show that RepairBoost can accelerate the repair by 35.0-97.1% for various erasure codes and repair algorithms.

1 Introduction

Today's storage systems are often composed of a large number of *storage nodes* (called "nodes" for brevity) to accommodate the explosively increasing data volume, making failures arise unexpectedly yet prevalently. To protect data reliability even in the presence of failures, many storage systems resort to *replication* [14] and *erasure coding* [47], both of which rely on pre-storing additional redundancy to repair the lost data. As opposed to replication that simply stores identical replicas, erasure coding can assuredly attain the same fault tolerance degree with much less storage consumption [59], and hence is much more preferable in commodity storage systems [4–6,41, 43]. In principle, erasure coding consists of two lightweight computational operations, namely *encoding* (i.e., generating redundant chunks based on the original data chunks) and *decoding* (i.e., repairing the original data chunks based on the surviving chunks), so as to realize the efficient transformation between the original data and redundancy.

Although being storage-efficient, erasure coding is prone to substantial *repair traffic* (i.e., the amount of data transmitted over the network for repair), as it often needs to retrieve multiple surviving chunks to repair a single chunk. To relieve the I/O amplification problem in repair, existing studies mainly resort to the following approaches: (i) constructing new theoretical erasure codes with provably reduced repair traffic (e.g., Locally Repairable Codes [21, 28, 52], Rotated-RS codes [27], and regenerating codes [13, 44, 49, 58]); (ii) designing efficient repair algorithms to parallelize the repair process [22, 32, 40, 56]; and (iii) utilizing machine-learning-based prediction techniques [16, 34, 42] to proactively restore the data with the repair algorithms before failure occurrence [38, 54] (see §2.2 for details).

Our observation is that most existing erasure codes and repair algorithms mainly focus on the single-chunk repair, yet the full-node failure (i.e., all the chunks in a node are permanently lost) must simultaneously manipulate the repair of multiple chunks. Thus, *there exists a gap between the deployment of existing repair approaches and the requirement of full-node repair*. Such a gap leads to several practical limitations: (i) they do not specifically leverage the full-duplex transmission to saturate the available bandwidth; (ii) they fail to carefully schedule the transmission of chunks to fully utilize the bandwidth at all times; (iii) they neglect the elastic cooperation of different repair algorithms to meet diverse reliability guarantees [25] and access popularities [9,23]; and (iv) they have dedicated repair strategies with the pre-specified data routing among nodes, thereby increasing the implementation complexity (see §2.3 for details). Therefore, how to seamlessly deploy existing repair approaches to efficiently tackle the full-node repair remains a challenging yet crucial issue in erasure-coded storage.

We bridge this gap by designing RepairBoost, a scheduling framework that can assist a variety of erasure codes and repair algorithms to speed up a full-node repair. The main idea behind RepairBoost is to formulate a single-chunk re-

*Corresponding author: Zhirong Shen (shenzr@xmu.edu.cn).

pair through a *repair directed acyclic graph* (RDAG), which characterizes the data routing over the network and the dependencies among the requested chunks for repair. RepairBoost then decomposes an RDAG into multiple *repair tasks*, each of which performs data uploads and downloads to facilitate the repair. RepairBoost balances the repair traffic and bandwidth utilization in the full-node repair through the following two steps: (i) it carefully dispatches the repair tasks of multiple RDAGs to the corresponding nodes for balancing the overall upload and download repair traffic; and (ii) it coordinates the execution orders of repair tasks to saturate the utilization of the available upload and download bandwidth. To summarize, our main contributions include:

- **[Design]** We propose RepairBoost to assist existing erasure codes and repair algorithms for speeding up the full-node repair. RepairBoost formulates a single-chunk repair through an RDAG. It then balances the upload and download repair traffic via carefully assigning the repair tasks of RDAGs to the nodes. RepairBoost also formulates the maximum flow problem [57] to schedule the data transmission for saturating the unoccupied bandwidths (§3.1-§3.3).
- **[Generality]** We also show that RepairBoost can be extended to tackle multiple node failures and boost the repair in heterogeneous environments (§3.4).
- **[Implementation]** We implement a prototype of RepairBoost with C++, which can be an independent middleware deployed atop existing storage systems for repair scheduling. We also demonstrate the portability of RepairBoost by integrating it into Hadoop HDFS 3.1.4 with limited modifications to the codebase (with around 270 LoC added) (§4).
- **[Evaluation]** We evaluate the performance of RepairBoost on Amazon EC2 [8], showing that it can support a variety of erasure codes and repair algorithms and increase the repair throughput by 35.0-97.1% (§5).

The source code of our RepairBoost prototype can be reached via **https://github.com/shenzr/repairboost-code**.

2 Background

We start with the basics of erasure coding (§2.1) and elaborate on existing attempts for repair acceleration in erasure-coded storage (§2.2). We also summarize the limitations that remain to be addressed (§2.3).

2.1 Basics

Erasure coding introduces slight computational operations to reduce the storage overhead for reliability assurance [59]. It operates on data in units of *chunks*, which are a collection of data with the size of several megabytes (e.g., 64 MB in Hadoop HDFS [6] and 256 MB in Facebook's data-warehouse cluster [48]). Formally, erasure coding can be configured via two integer parameters, namely k and m, which tune the storage efficiency and fault tolerance assurance. A (k,m) erasure code encodes every k equal-sized data chunks $\{D_1, D_2, \cdots, D_k\}$ at a time to generate additional m redundant chunks (called *parity chunks*) $\{P_{k+1}, P_{k+2}, \cdots, P_{k+m}\}$. In this paper, we mainly consider the *linear erasure codes*, where each parity chunk can be expressed as a *linear combination* of the k data chunks via the Galois Field arithmetic [46], given by $P_{k+j} = \sum_{i=1}^{k} \alpha_{i,j} D_i$, where $\alpha_{i,j}$ $(1 \leq i \leq k$ and $1 \leq j \leq m)$ is the encoding coefficient used by the data chunk D_i to calculate the parity chunk P_{k+j}. Such $k+m$ data and parity chunks that are encoded together collectively constitute a *stripe* (or a coding group [33, 62]), ensuring that *any* k out of $k+m$ chunks of the same stripe *always* suffice to restore (decode) the original k data chunks (a.k.a. *Maximum Distance Separable* (MDS) property in coding theory). In other words, the (k,m) erasure code can tolerate *any* m chunk failures for every stripe.

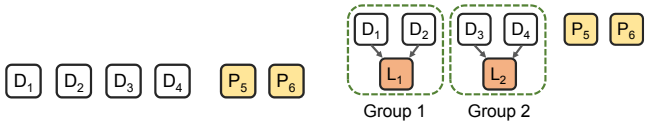
In practice, data chunks in storage systems are often organized into multiple stripes, which are *independently* manipulated by erasure coding. Hence, by distributing the $k+m$ chunks of each stripe across $k+m$ distinct nodes (i.e., one chunk per node), we can always guarantee data reliability in the face of no more than *any* m node failures. In this paper, we equally treat the data and parity chunks in repair, and just refer to them as "chunks."

A variety of erasure codes have been proposed for decades, where Reed-Solomon (RS) codes [51] are the most popular code construction used in production storage (e.g., Ceph [4], Hadoop HDFS [6], QFS [43], Facebook f4 [41], and Yahoo Cloud Object Store [5]), since they support any parameters of k and m. For simplicity, we denote the RS code configured by the parameters k and m as $\mathrm{RS}(k,m)$ and use it as our case study throughout the paper. For example, Figure 1(a) depicts a stripe of $\mathrm{RS}(4,2)$. We also show that our work is applicable to other linear erasure codes (§5.3).

2.2 Repair

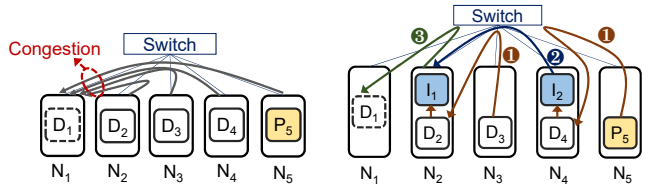
Repair in erasure-coded storage is often classified into (i) *full-node repair*, which restores all lost chunks in a failed node (e.g., caused by a disk failure), and (ii) *degraded reads* to the temporarily unavailable chunks (e.g., caused by system upgrades and network disconnections) or the chunks that have not yet been repaired in the full-node repair. In this paper, we mainly focus on the *single full-node repair*, which is recognized as one of the top causes of service downtime [35]. Our work can also tackle multiple node failures (§3.4).

The full-node repair has to manipulate the repair across multiple stripes. Although being storage-efficient, erasure coding is prone to a high repair penalty. For example, $\mathrm{RS}(k,m)$ needs to retrieve k surviving chunks of the same stripe to recover a lost chunk, thereby amplifying the storage and network I/Os in repair by k times. Specifically, suppose that a chunk C^* fails and $\{C_1, C_2, \cdots, C_k\}$ is the set of k chunks



(a) A stripe of RS(4,2). (b) A stripe of LRC(4,2,2).

Figure 1: Examples of RS codes and LRCs.



(a) A repair solution of conventional repair (CR)

(b) A repair solution of PPR [40]

Figure 2: Repair solutions for RS$(4,2)$. D_1 indicates the repaired chunk. N_1 is the destination node.

within the same stripe of C^*. We can repair C^* using a linear combination of the k chunks as:

$$C^* = \sum_{i=1}^{k} \beta_i C_i, \tag{1}$$

where β_i is the decoding coefficient used by C_i for repairing C^*. In this paper, we refer to the node storing the repaired chunk as the *destination node*.

In view of the high repair penalty, existing studies design solutions that mitigate the repair penalty as follows.

Repair-efficient erasure codes: They relieve the repair traffic with theoretical guarantees through the construction of erasure codes, where *Locally Repairable Codes* (LRCs) [21, 52] and regenerating codes [13, 44, 48, 49, 58] are two representatives. LRCs are configured via three parameters, namely k, l, and m. LRC(k,l,m) extends RS(k,m) by organizing the k data chunks of a stripe into l *groups* (assuming that k is divisible by l) and maintaining an additional *local parity chunk* for each group. This design allows a single data chunk to be repairable by retrieving merely $\frac{k}{l}$ surviving chunks within the same group[1]. Figure 1(b) shows a stripe of LRC$(4,2,2)$, where L_i is the local parity chunk of the i-th group $(1 \leq i \leq 2)$.

On the other hand, regenerating codes reduce the repair traffic by (i) requiring surviving nodes to send the linear combination of the locally stored data [13] and (ii) contacting more surviving nodes to assist the repair [49]. Recently proposed regenerating codes (e.g., Butterfly codes [44] and Clay codes [58]) further eliminate the needs of computations in the surviving nodes, which can simply retrieve the required sub-chunks directly from surviving nodes for repair.

Repair algorithms: While erasure codes specify *which chunks* should be retrieved for repair, repair algorithms mainly focus on *how* to quickly retrieve the surviving chunks over the network for existing erasure codes. They accelerate the repair by exploiting the unoccupied bandwidth without reducing the repair traffic. Each repair algorithm specifies a *repair solution* for a lost chunk, including the data routing and execution orders among the surviving nodes. Figure 2(a) shows the repair solution of D_1 under the *conventional repair* (CR) for RS$(4,2)$, where it transmits four surviving chunks $\{D_2, D_3, D_4, P_5\}$ at the same time to the destination node N_1. Suppose that a chunk can be transmitted over a network link in a *timeslot*. The conventional repair takes four timeslots for N_1 to download the four chunks, because of the congestion of the download link of N_1. PPR [40] relieves the network congestion by decomposing a single-chunk repair into multiple sub-stages and executing them in parallel, so as to fully utilize the available bandwidth. Figure 2(b) depicts a single-chunk repair solution under PPR for RS$(4,2)$, which accomplishes the repair in three timeslots: (i) in the first timeslot (❶), we transmit D_3 to D_2 and add them together (as in Equation (1)) to generate an intermediate chunk I_1, while at the same time sending P_5 to D_4 to generate another intermediate chunk I_2; (ii) in the second timeslot (❷), we transmit I_2 to I_1 and add I_2 with I_1 to restore the lost chunk D_1 (as in Equation (1)); and (iii) in the third timeslot (❸), we deliver D_1 to the destination node. Thus, PPR exploits the available bandwidth across multiple nodes at each timeslot to mitigate the network congestion in a single-chunk repair. ECPipe [32] further reduces the repair time to almost one timeslot by decomposing the repair of a lost chunk into the pipelined repair operations of multiple smaller sub-chunks.

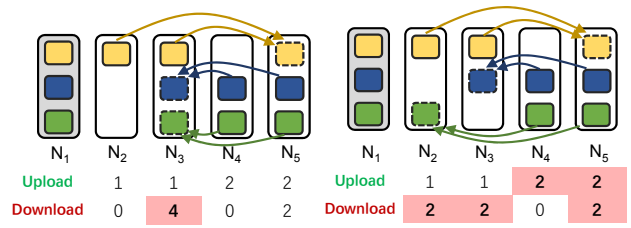
Proactive repair with erasure coding: Both repair-efficient erasure codes and repair algorithms are *reactive*, implying that they launch the repair operation only *after* failures truly occur. *Proactive* repair [38, 54] further reduces the window of data vulnerability by using *machine-learning-based failure prediction models* (e.g., Bayes classifier [16], support vector machines [42], and random forest [34]), such that it can detect in advance the nodes impending to fail and *proactively* repair the data in such nodes *before* the failure occurrence. Proactive repair can be realized by the data migration and the decoding of chunks based on erasure coding [54], where we mainly focus on the latter in this paper.

2.3 Limitations

By examining existing attempts to accelerate the repair of erasure-coded storage, we identify the following limitations that, if not properly addressed, will limit the performance gains when they are directly employed for the full-node repair.

Limitation 1 (L1): Failing to utilize the full duplex transmission. Most existing studies overlook the *full duplex transmission* [12, 36] in repair, which enables a node to send (upload) and receive (download) data simultaneously and independently; as a result, they cannot properly balance the upload and download repair traffic in the full-node repair. Figure 3 shows two examples of repair under CR (where N_i is the i-th

[1] LRC(k,l,m) also needs k chunks to repair a global parity chunk (e.g., P_5 and P_6 in Figure 1(b)).



(a) Unbalanced repair solutions (b) Balanced repair solutions

Figure 3: L1 (Failing to utilize the full duplex transmission). The repair time is determined by the most loaded node (with the red color in the background).

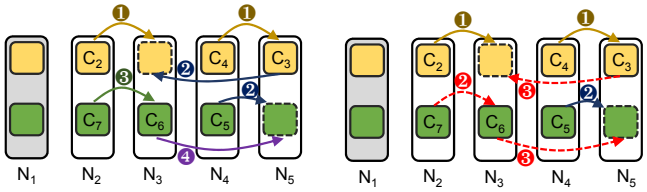
node), where the first example needs to download four chunks (at N_3) to accomplish the repair, while the latter only uploads and downloads at most two chunks for repair. This example indicates that balancing the upload and download repair traffic in the full duplex transmission has the potential to reduce the overall repair time, which is determined by the node with the most upload or download repair traffic.

Specifically, the repair-efficient codes [13,21,44,48,49,58] reduce a single chunk's repair traffic without concerning traffic balancing. Some repair algorithms (e.g., ECPipe [32] and PPR [40]) mainly focus on accelerating a single chunk's repair, while paying limited attention to balance the upload and download repair traffic in the full-node repair. Although ClusterSR [55] balances the inter-cluster upload and download repair traffic, it still does not balance the upload and download bandwidths in general storage systems.

Limitation 2 (L2): Failing to fully utilize the bandwidth at each timeslot. While existing repair algorithms can relieve the download bottleneck within a single-chunk repair (e.g., PPR [40] and ECPipe [32]), they simply combine the repair solutions of multiple chunks to cope with the full-node repair. This may unintentionally lead to the link congestion again and make the bandwidth under-utilized in the full-node repair.

Figure 4 shows how transmission scheduling affects the repair. In Figure 4(a), after the transmissions of C_2 and C_4 in the first timeslot, the transmissions of C_3 (from the node N_5 to N_3) and C_7 (from N_2 to N_3) compete for the bandwidth of the download link of N_3. Figure 4(a) gives priority to C_3 in the second timeslot. Since C_6 can only be transmitted after receiving C_7, Figure 4(a) finally needs four timeslots. As a comparison, Figure 4(b) sends C_7 at the second timeslot, allowing the transmission of C_6 and C_3 to be performed in parallel without bandwidth competition. Hence, Figure 4(b) only uses three timeslots. This example indicates that the repair solutions in the full-node repair should be carefully scheduled.

Limitation 3 (L3): Inflexibility. Many repair algorithms [22, 32, 40, 55, 56] treat every chunk equally and repair all the lost chunks using the same repair algorithm. This repair fashion is simple yet inflexible, as it cannot elastically combine different repair algorithms, and make them cooperate for diverse reliability requirements [25] and skewed access popularity in real-world storage systems [9, 23]. For example, we can combine ECPipe [32] and CR, such that we use ECPipe to repair the chunks that require higher reliability guarantee, while employing CR to restore the remaining ones, so as to regulate the repair traffic and foreground traffic.



(a) Repair using four timeslots (b) Repair using three timeslots

Figure 4: L2 (Failing to fully utilize the bandwidth at each timeslot). The transmission scheduling affects the bandwidth utilization.

Limitation 4 (L4): Lack of a general framework for the full-node repair. Existing distributed storage systems deploy different specific repair algorithms for a single-chunk repair. For example, many commodity storage systems (e.g., Hadoop HDFS [7] and Windows Azure Storage [21]) still rely on CR in a single-chunk repair due to its simplicity, while PPR [40] and CAR [56] can be used in hierarchical storage systems to reduce inter-rack repair traffic. It is desirable to have a *general* framework that simultaneously supports different types of repair algorithms for different deployment scenarios.

3 RepairBoost Design

We present RepairBoost, a scheduling framework that can assist a variety of erasure codes and repair algorithms to boost the full-node repair performance.

Assumptions: RepairBoost is designed based on the following assumptions. First, RepairBoost mainly focuses on a single full-node repair, which is reported as the most predominant failure event in practice (e.g., 98% of the total failure events [50]). Nevertheless, RepairBoost can be extended to tackle a multi-node failure (i.e., more than one node in a stripe fails) (§3.4). Second, to simplify our discussion, we elaborate RepairBoost using RS codes in homogeneous environments (i.e., with identical link bandwidth), yet we also show that RepairBoost works for other erasure codes (§5.3) and can be deployed in heterogeneous environments (i.e., with different link bandwidth) (§3.4).

Overview: RepairBoost uses the following techniques to address the limitations aforementioned (§2.3). It first abstracts a single-chunk repair solution via a general *repair directed acyclic graph (RDAG)* (§3.1). By supporting the scheduling of multiple RDAGs, RepairBoost can achieve both *flexibility* (i.e., allowing the collaboration of different repair algorithms; L3 addressed) and *generality* (i.e., being workable for various erasure codes and repair algorithms; L4 addressed).

RepairBoost then decomposes multiple RDAGs into vertices that have dedicated repair tasks. It then carefully assigns

the repair tasks to the nodes, so as to balance the overall upload and download repair traffic across the surviving nodes (§3.2; L1 addressed).

RepairBoost finally constructs a directed network based on the surviving nodes and the corresponding repair tasks. It then determines the chunks to be transmitted by solving a maximum flow problem [57], so as to fully saturate the unoccupied upload and download bandwidth at each timeslot (§3.3; L2 addressed).

3.1 Repair Abstraction

RDAG construction: We first formalize a single-chunk repair solution through a *directed acyclic graph* (DAG), which is called the *Repair DAG* (RDAG). For RS(k,m), the RDAG of a lost chunk can be initialized over $k+1$ vertices, where the k vertices $\{v_1, v_2, \cdots, v_k\}$ represent the k nodes that store the requested surviving chunks for repair, while v_{k+1} denotes the destination node. Also, we employ directed edges among vertices to represent the data routing directions specified in repair algorithms.

We construct the edges based on the following rules. Given two vertices v_i and v_j $(1 \le i \ne j \le k+1)$, we use a directed edge $e_{i,j}$ to indicate that v_i is designated to send a surviving chunk to v_j. Hence, if $e_{i,j}$ exists, we say that v_i is a *child* of v_j, and conversely v_j is the *parent* of v_i. For v_j (where $j \ne k+1$), it has to collect all the requested surviving chunks from its children, add them together with the local data it stores using the decoding coefficient (see Equation (1)), and send the result to its parent. Therefore, in this RDAG, v_j can send a chunk for repair once all the chunks required from its children are received. As the full-node repair has to recover multiple chunks, a node may have multiple parents and children across different RDAGs.

Repair process guided by RDAG: The repair starts from the *leaf vertices* (i.e., those that do not have any child) and ends at v_{k+1}: for each edge $e_{i,j}$ $(1 \le i \ne j \le k+1)$, if v_i has sent the requested chunk to v_j for repair, we remove $e_{i,j}$ from the RDAG; further, for each leaf vertex v_i, if there is no edge connecting to it (indicating that v_i has already transmitted all the requested chunks to its parents), we remove v_i from the RDAG as well. Hence, as the repair proceeds, the number of vertices in the RDAG will gradually decrease and the lost chunk is successfully repaired once the vertex v_{k+1} becomes a leaf vertex eventually.

Example: We take the RDAG of PPR in Figure 5 as an example, where $k=4$. Suppose that the four requested surviving chunks in $\{v_1, v_2, \cdots, v_4\}$ are denoted by $\{C_1, C_2, \cdots, C_4\}$, respectively. In the first stage, v_1 sends a chunk $\beta_1 C_1$ to its parent v_2 (where β_1 is the decoding coefficient of C_1; see Equation (1)), while at the same time v_3 sends another chunk $\beta_3 C_3$ to its parent v_4. We also update the RDAG by removing $e_{1,2}$, $e_{3,4}$, v_1, and v_3. In the second stage, the leaf vertex v_2 combines $\beta_1 C_1$ with its stored chunk C_2 and sends the result $\beta_1 C_1 + \beta_2 C_2$ to its parent v_4. In the third stage, after collecting two chunks from its children, v_4 combines $\beta_1 C_1 + \beta_2 C_2$, $\beta_3 C_3$, and its locally stored chunk C_4 to restore the lost chunk $C^* = \sum_{i=1}^{4} \beta_i C_i$, and finally sends C^* to v_5. The repair of C^* completes when v_5 becomes a leaf vertex.

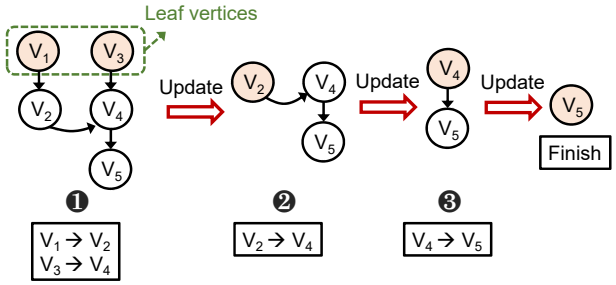


Figure 5: Example of an RDAG of PPR [40] when $k=4$. The vertex with pink color indicates the leaf vertex. We repeatedly update an RDAG as the repair proceeds.

Advantages of an RDAG: The RDAG design has the following advantages. First, an RDAG is a general formalization of a single-chunk repair solution. That is, we can construct an RDAG for any given repair algorithm once we identify the surviving chunks needed for repair and the data routing strategy performed among them. Thus, RepairBoost is applicable to a variety of erasure codes (e.g., RS codes, LRCs, and regenerating codes) and single-stripe repair algorithms (e.g., CR, PPR, and ECPipe). This design also resolves the tensions of deploying specific repair algorithms (i.e., being general) and facilitates their co-existence (i.e., being flexible).

Second, an RDAG describes how data is transmitted over the network and also the dependency (or relationship) among the k surviving chunks involved in the repair. Given an RDAG, we can readily identify that the leaf vertices with different parents in an RDAG can be potentially transmitted in parallel to occupy the available bandwidth without network congestion (e.g., v_1 and v_3 in Figure 5). We leverage this property in the transmission scheduling (§3.3).

Third, an RDAG also indicates the repair tasks of each vertex. For example, we can learn that v_4 in Figure 5 needs to download two chunks and upload one chunk in the repair. We explore the traffic balancing through the assignment of repair tasks (§3.2).

Discussion: OpenEC [29] proposes an ECDAG to characterize the encoding and decoding processes. Both an RDAG and an ECDAG are fundamentally different in the following aspects: (i) *physical meanings of edges*: the edges in an ECDAG represent the encoding and decoding operations, while the edges in an RDAG specify the routing and dependencies in repair; (ii) *graph structures*: an ECDAG introduces *virtual vertices* to denote the intermediate chunks generated, while an RDAG is built over the chunk repaired and the surviving chunks for repair; and (iii) *graph maintenance*: an ECDAG keeps the graph unchanged throughout the encoding and decoding processes, while an RDAG is iteratively updated with the repair progress.

Algorithm 1 Mapping of Intermediate Vertices

Input: $\mathcal{T}$ (set of intermediate vertices)
Output: The mapping $\mathcal{M}$ from intermediate vertices to nodes

```
1:  procedure MAIN(T)
2:      Set M = ϕ
3:      Sort T in descending order
4:      while T ≠ ϕ do
5:          v̄ = POP(T)
6:          Establish N
7:          N* = MAP(v̄, N)
8:          Append (v̄, N*) to M
9:      end while
10:     return M
11: end procedure
12: function MAP(v̄, N)
13:     Set N* = arg min{d_{N_i} | N_i ∈ N}
14:     T = T − v̄
15:     u_{N*} = u_{N*} + u_{v̄}
16:     d_{N*} = d_{N*} + d_{v̄}
17:     return N*
18: end function
```

3.2 Repair Traffic Balancing

After constructing the RDAGs for the lost chunks in a failed node, RepairBoost then assigns the repair tasks by mapping the vertices to the nodes, such that (i) the fault tolerance degree (i.e., the tolerable number of failed nodes) offered by erasure coding must be preserved after repair, and (ii) the upload and download repair traffic across the whole system should be as balanced as possible.

Retaining fault tolerance degree: Given an RDAG of a lost chunk, we assign the k vertices $\{v_1, v_2, \cdots, v_k\}$ to the k nodes that store the surviving chunks of the same stripe. We also map v_{k+1} to the node that does not store any chunk of the same stripe before the repair. By doing so, RepairBoost ensures that the $k+m$ chunks of the same stripe still reside in $k+m$ different nodes after repair, thereby retaining the node-level fault tolerance offered by erasure coding (§2.1).

Balancing repair traffic: Given an RDAG, we represent the repair task of a vertex v_i by a tuple (u_{v_i}, d_{v_i}) (where $1 \leq i \leq k+1$), which indicates the numbers of chunks being uploaded and downloaded by v_i in this RDAG, respectively.

We classify the vertices of an RDAG into three categories: (i) the leaf vertex v_i, which only sends (uploads) surviving chunks for repair (i.e., $u_{v_i} > 0$ and $d_{v_i} = 0$); (ii) the *root vertex*, which only receives (downloads) data for repair (i.e., $u_{v_i} = 0$ and $d_{v_i} > 0$); and (iii) the intermediate vertex, which receives (downloads) multiple chunks from its children and sends (uploads) one intermediate chunk to its parent (i.e., $u_{v_i} = 1$ and $d_{v_i} > 0$). For example, in Figure 5, v_1 is a leaf vertex, v_2 is an intermediate vertex, and v_5 is the root vertex.

After collecting the three kinds of vertices from multiple RDAGs, our main idea is to give priority to map the intermediate and root vertices to the nodes, with the primary objective of balancing the download repair traffic at first. We then carefully assign the leaf vertices to further balance the upload repair traffic. Algorithm 1 shows the pseudo-code of the mapping algorithm for the intermediate vertices.

Algorithm details: Let $\mathcal{T}$ be the set of the intermediate vertices that have not been assigned yet. Let $\mathcal{M}$ be the mapping established. RepairBoost first initializes $\mathcal{M}$ as an empty set and sorts the intermediate vertices in $\mathcal{T}$ by the number of chunks required to be downloaded in descending order (Lines 2-3 in Algorithm 1). It then pinpoints the vertex $\bar{v}$ that needs to download the most chunks for repair among the vertices of $\mathcal{T}$ (Line 5). RepairBoost finds the set of nodes (denoted by $\mathcal{N}$) that store the surviving chunks requested in the RDAG of $\bar{v}$ but have not been assigned the repair task of this RDAG (Line 6). It then searches the node for $\bar{v}$ by calling the MAP function (Line 7).

In the MAP function, RepairBoost selects the node N^* that has the least download traffic among the ones in $\mathcal{N}$ (Line 13, where d_{N_i} denotes the number of downloaded chunks of the node N_i). It then maps $\bar{v}$ to N^*. RepairBoost later excludes $\bar{v}$ from the set of $\mathcal{T}$ (Line 14), and increases the numbers of the uploaded and downloaded chunks (denoted by u_{N^*} and d_{N^*}) of N^* by the task of $\bar{v}$ (Lines 15-16). The mapping indicates that N^* will serve as $\bar{v}$ in the corresponding RDAG by uploading and downloading the requested surviving chunks specified to $\bar{v}$. RepairBoost repeats the above steps until all the intermediate vertices have been assigned to the corresponding nodes (Lines 4-9).

We analyze the computational complexity of Algorithm 1. Suppose that the number of nodes participating in repair is n and $|\mathcal{T}|$ is the number of intermediate vertices in $\mathcal{T}$. We note that: (i) the sorting of the intermediate vertices (Line 3) incurs a computational complexity of $O(|\mathcal{T}| \cdot \log(|\mathcal{T}|))$, and (ii) the mapping of an intermediate vertex (Lines 5-8) incurs a computational complexity of $O(n \cdot \log n)$, which will be executed for $|\mathcal{T}|$ times (Line 4). Thus, the overall computational complexity of Algorithm 1 is $O(|\mathcal{T}| \cdot \log(|\mathcal{T}|) + |\mathcal{T}| \cdot n \cdot \log n)$.

RepairBoost then maps the root and leaf vertices to the corresponding nodes in the similar way, except the following differences: (i) when mapping the root vertex of an RDAG, RepairBoost selects the node with the lightest download repair traffic among the ones, which does not store any chunk within the same stripe of the repaired chunk; (ii) when mapping the leaf vertices of an RDAG, RepairBoost chooses the nodes with the lightest upload repair traffic among the ones, which not only store the surviving chunks within the same stripe of the repaired chunk, but also have not been mapped with any vertex of this RDAG.

Example: Figure 6 shows an example of mapping an RDAG to the nodes. Given an RDAG, we decompose it into vertices with the tuples (u_{v_i}, d_{v_i}) ($1 \leq i \leq k+1$ and $k = 4$ in this example), which specify the numbers of uploaded and downloaded chunks for repair (Step ❶). For example, v_2 needs to upload and download one chunk. To map the intermedi-

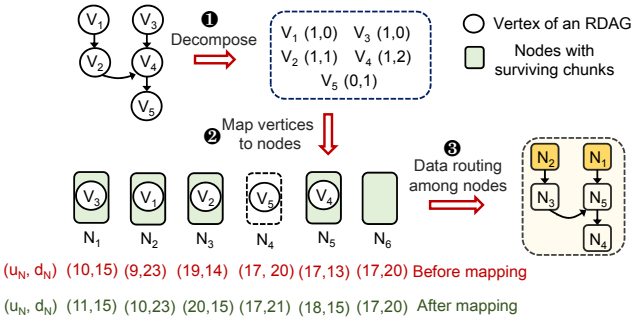


Figure 6: Example of mapping vertices of an RDAG to nodes. We select an RDAG of PPR to repair a chunk of RS$(4,2)$. N_2 and N_1 are the leaf vertices of this RDAG after mapping. The numbers in red (resp. green) denotes the upload and download traffics afforded by a node before (resp. after) the mapping.

ate vertices (i.e., v_2 and v_4), we first consider v_4 as it needs to download the most chunks. Since N_5 needs to download the fewest chunks (i.e., 13) among $\{N_1,N_2,N_3,N_5,N_6\}$ that store the surviving chunks, we map v_4 to N_5, and update the numbers of uploaded and downloaded chunks afforded by N_5 afterwards (Step ❷). After all the five vertices of an RDAG have been assigned, we can learn the data routings among nodes $\{N_1,N_2,\cdots,N_5\}$ to repair the chunk (Step ❸).

3.3 Transmission Scheduling

After establishing the mapping from the RDAGs to the nodes for balancing the overall upload and download repair traffic (§3.2), it does not necessarily achieve the lower-bound of the repair time, as the bandwidth may not be utilized at each timeslot during the repair (L2 in §2.3).

To further saturate the bandwidth utilization, RepairBoost formulates the transmission scheduling as a *maximum flow problem* [57]. Specifically, suppose that there are n nodes participating in the full-node repair. We can construct a network based on the RDAGs of the lost chunks. This network is built over $2n+2$ vertices (Figure 7), with a source s, a sink t, n sender vertices $\{S_1,S_2,\cdots,S_n\}$ representing the n nodes that can potentially send data for repair, and another n receiver vertices $\{R_1,R_2,\cdots,R_n\}$ representing the n nodes that may receive data at the same time. We then establish the connections as follows: for any two vertices S_i and R_j $(1 \le i \ne j \le n)$, we establish the connection between S_i and R_j once S_i can send a chunk to R_j according to the RDAGs. Each connection between S_i and R_j is assigned with the capacity of one, implying that we can send one surviving chunk at a time from S_i and R_j. Thus, our objective is to find a maximum flow over the network whose capacity denotes the most chunks that can be transmitted simultaneously at this timeslot, so as to saturate the available upload and download bandwidth.

After establishing the maximum flow, we dispatch the chunks according to the selected edges of the maximum flow. If S_i has many chunks to be sent, we prefer to send a chunk, such that sending this chunk can make a parent of S_i become a leaf vertex in an RDAG in the next timeslot (§3.1). Our

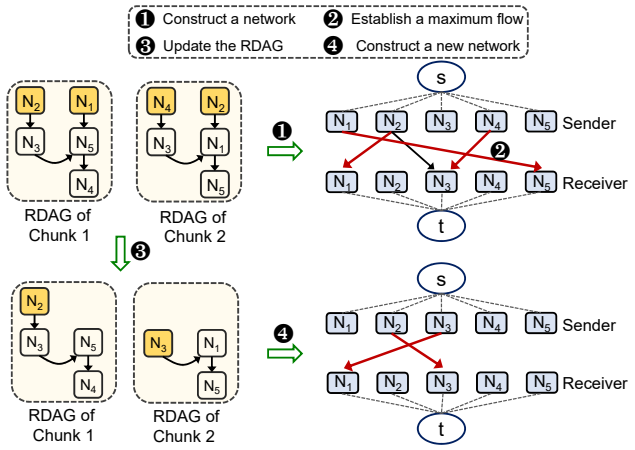


Figure 7: Example of transmission scheduling (where $k=4$ and $n=5$). The red arrows means the maximum flows established.

objective is to establish the maximum flow, while at the same time helping increase the number of edges of the network in the next scheduling. This can potentially increase the capacity of the maximum flow in the next transmission.

Once the chunks associated with the edges of the maximum flow have all been transmitted, we accordingly delete the corresponding edges in the RDAGs and update the network based on the residual RDAGs. We repeat the scheduling until all the lost chunks of the failed node have been repaired.

Example: Figure 7 shows an example to repair two chunks among five surviving nodes (i.e., $n=5$). There are three nodes (i.e., N_1, N_2, and N_4) that can send chunks for repair (marked in yellow color) in the RDAGs at the very beginning. We construct the network (Step ❶) based on the two RDAGs and find the maximum flow (Step ❷). The maximum flow indicates that we can send the chunks as follows for transmitting the most chunks simultaneously: $N_1 \to N_5$ (in the RDAG of the chunk 1), $N_2 \to N_1$ (in the RDAG of the chunk 2), and $N_4 \to N_3$ (in the RDAG of the chunk 2). We then update the RDAGs by removing the edges associated with the transmitted chunks and marking the leaf vertex at this time (Step ❸). We repeat the construction of the network and the finding of the maximum flow for the residual RDAGs (Step ❹).

Complexity analysis: Suppose that there are n nodes and e edges in a network. We can use Dinic's algorithm [24] to find the maximum flow, whose computational complexity is $O(n^2e)$.

3.4 Extensions

Multi-node repair: We offer two options when using RepairBoost to cope with multi-node repair. The first approach is to simply repair each failed node individually until all the failed nodes are repaired successfully. The second approach is to give priority to repairing the stripes that comprise more failed chunks or popularly accessed chunks, so as to meet the requirements on system reliability and access performance.

Heterogeneous environments: RepairBoost can also be adapted to heterogeneous environments. When given the available link bandwidth, RepairBoost can deduce the time to upload and download a chunk at first. It can amend the repair traffic balancing (§3.2) by mapping the vertices to the nodes based on the times for uploading and downloading a chunk (rather than the numbers of uploaded and downloaded chunks in the design for homogeneous environment). This can ensure that the upload and download times are almost the same across the nodes.

RepairBoost then adopts the following ways in the polling mode for transmission scheduling: (i) it pinpoints the node (denoted by N') that spends the least time in uploading and downloading data for repair; (ii) it sorts the set of links $\mathcal{L}$ connected to N' based on the available bandwidth; and (iii) it picks the link from $\mathcal{L}$ that owns the largest upload or download bandwidth for data transmission. RepairBoost repeats the above steps until all the requested chunks are transmitted.

Adaptation to network conditions: RepairBoost can also adapt to network conditions (e.g., random network congestion). Specifically, RepairBoost can break an entire full-node repair process into several subprocesses, each of which repairs a number of single chunks. RepairBoost can then monitor the completion times of nodes in each subprocess to infer the network status, and proactively adjust the repair solutions of next subprocesses. For the node that takes the longest (resp. shortest) time in a subprocess, we can speculatively lessen (resp. increase) the upload and download repair traffic it affords in the next subprocess to balance the repair time across the surviving nodes.

4 Implementation

We implement a prototype of RepairBoost in C++ with around 3,200 lines of codes (LoC), which can serve as an independent middleware running atop existing storage systems to instruct the repair operations on behalf of them. We realize the encoding and decoding functionalities based on the coding library Jerasure v2.0 [2]. We maintain an in-memory key-value store in each node and transmit data through the interfaces of Redis [3].

System architecture: Figure 8 presents the architecture of RepairBoost, which comprises a *coordinator* sitting on the metadata server and multiple *agents* running on the nodes (with one agent per node). The coordinator manages the metadata of stripes (e.g., the mapping from chunks to stripes, and the nodes that the $k+m$ chunks of every stripe reside), while the agents are standby to wait for the repair commands and perform the repair operations cooperatively.

Operating flow: Once a node failure event is reported to the metadata server, the coordinator first pinpoints the IDs of the lost chunks as well as the identities of the associated stripes. It then establishes the *repair solution* of each lost chunk,

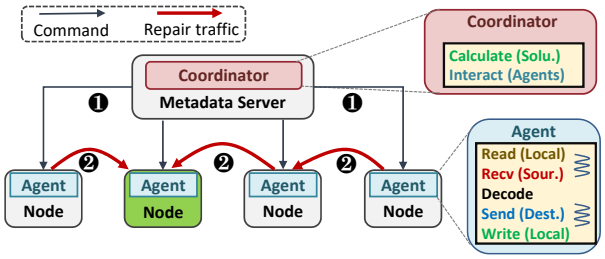


Figure 8: System architecture of RepairBoost. For simplicity, we only illustrate the repair of a single stripe. The node with green color denotes the destination node of this stripe.

including the k surviving chunks selected to participate in the repair, the data routing among the nodes, and the destination node to store the repaired chunk. The repair solutions are sealed into *repair commands* whose format is pre-determined and understood by the agents, which will then be sent to the agents that involve in the repair operations (Step ❶).

Upon receiving the repair commands, the agents first extract the repair solutions. They then work cooperatively by (i) reading the requested surviving chunks locally stored, (ii) sending them to the appointed relay nodes, and (iii) decoding (repairing) the lost chunks and storing them locally (Step ❷). To accelerate the repair operations, RepairBoost partitions a chunk into many smaller *packets* and uses the multi-threading technique to pipeline the disk I/O, network transmission, and computation in unit of packets.

Integration with Hadoop HDFS: We also integrate RepairBoost into Hadoop HDFS 3.1.4 [6] by adding around 270 LoC [2]. We deploy the coordinator in the NameNode and run agents in the DataNodes. Specifically, when the NameNode is aware of the node failure through periodical heartbeats issued by the DataNodes, the coordinator intercepts the repair commands and calculates the repair solutions via extracting the metadata information (about the addressing of stripes and chunks) from the NameNode [3]. We then trigger RepairBoost to repair the lost data on behalf of HDFS. To facilitate the integration, RepairBoost also adds a `RepairBoostReconstructor` class in the `erasurecode` package of the DataNode, which collects the repaired chunks from its agent and writes them to the underlying storage of HDFS.

5 Performance Evaluation

We carry out extensive testbed experiments to evaluate the performance of RepairBoost. We summarize our major findings as follows: (i) RepairBoost can improve the repair throughput by 35.0-97.1% (§5.2-§5.4); (ii) RepairBoost can assist the

[2] Although Hadoop HDFS 3.1.4 employs Intel ISA-L [1] for erasure coding realizations, we can still employ Jerasure Library to repair the lost data with the same Cauchy matrix [2]. We have confirmed the decoding correctness in our evaluation.

[3] RepairBoost accesses the metadata of the Namenode via executing the command "`hdfs fsck / -files -blocks -locations`".

repair for a variety of erasure codes and repair algorithms (§5.3); (iii) RepairBoost is more advantageous in the environment with lower network bandwidth (§5.2); (iv) RepairBoost retains its effectiveness when being used in heterogeneous environments and multi-failure repair (§5.4).

5.1 Setup

Testbed and preparations: We evaluate the performance of RepairBoost on Amazon EC2 [8] to unveil its performance in a real-world cloud scenario. We set up 17 virtual machine instances with the type of `m5.large` in the US East (North Virginia) region. Each instance runs Ubuntu 16.04.7 LTS, and is equipped with two vCPUs with 2.5 GHz Intel Xeon Platinum, 8 GB RAM, and 40 GB of EBS storage. The network bandwidth between any two instances is around 1 Gb/s (measured by `iperf`) and the disk bandwidth is around 130 MB/s. Among the 17 instances, we run the RepairBoost coordinator on one instance and deploy the RepairBoost agents on the remaining 16 instances.

Before triggering the repair, we first warm up the system by writing sufficient data encoded by the selected erasure coding scheme. We then erase the data of one instance to mimic a node failure and launch RepairBoost for data repair. We measure the latency, from the time when the failure event is reported to the time when all the lost data is repaired and persisted. We mainly focus on the *repair throughput*, defined as the size of data repaired per time unit. A higher repair throughput indicates a shorter window of vulnerability and hence stronger data reliability.

Erasure codes and repair algorithms: We demonstrate the flexibility, generality, and effectiveness of RepairBoost via deploying RepairBoost atop the following representative erasure codes and repair algorithms. We mainly consider three representative erasure codes: (i) the conventional RS codes [51] that are popularly used in today's storage systems; (ii) LRCs [21, 52] that trade additional storage overhead for reducing the repair traffic, and (iii) Butterfly code [44], a systematic *minimum storage regenerating* (MSR) code with the minimum repair traffic for a single-node repair.

We also focus on three typical repair algorithms: (i) the conventional repair that transmits k chunks directly to the destination node for repair; (ii) PPR [40], which decomposes a single-chunk repair into sub-stages and exploits the execution parallelism of the sub-stages; and (iii) ECPipe [32], which partitions a chunk into equal-sized slices and pipelines their transmission across the surviving nodes for repair.

Selection of baseline: To select an appropriate baseline approach for comparison, we consider two candidates: a *random selection* (RAN) approach and an LRU-based selection approach. Specifically, the random selection randomly chooses k nodes for data retrieval from the $k+m-1$ nodes that store the surviving chunks of the corresponding stripe, and also a destination node for data repair from the remaining nodes (§3.2). In the LRU-based selection, we track the timestamp of each node when it was last selected for repair, and choose the k nodes for data retrieval from the $k+m-1$ corresponding nodes and a destination one for data repair with the smallest timestamps. When the nodes to be selected have the same timestamp, the LRU-based selection always chooses the one with the smallest node ID.

Code	CR		ECPipe		PPR	
	RAN	LRU	RAN	LRU	RAN	LRU
RS(6,3)	**1.64**	1.65	**1.25**	1.44	**1.30**	1.43
RS(10,4)	**1.67**	1.82	**1.14**	1.41	**1.25**	2.01
LRC(6,2,2)	**1.67**	1.68	1.34	**1.32**	1.46	**1.37**
LRC(8,2,2)	**1.63**	1.65	**1.25**	1.26	**1.33**	1.43
RC(4,2,5)	**1.71**	1.74	**1.29**	1.31	1.37	**1.32**
RC(5,3,6)	**1.70**	1.73	**1.25**	1.30	**1.33**	1.35

Table 1: Comparison of the random selection and LRU-based selection in terms of their load balancing degrees.

We evaluate the *load balancing degrees* of the two approaches as $\frac{L_{max}}{L_{balanced}}$, where L_{max} represents the maximum upload and download repair traffic of a node across the system, while $L_{balanced}$ denotes the upload (or download) repair traffic that is balanced on a node. We can determine $L_{balanced}$ by dividing the amount of repair traffic by the number of surviving nodes.

We consider two representative configurations for each erasure code: RS(6,3) (also used in QFS [43] and Hadoop HDFS [6]), RS(10,4) (used in Facebook f4 [41]), LRC(6,2,2) (also deployed in Windows Azure Storage [21]), LRC(8,2,2) [28], RC(4,2,5) [4] [44] and RC(5,3,6) (a high-rate MSR code that requires $d = k+1$ [53]). We randomly distribute the chunks of a stripe across 16 nodes and repeat the test for five runs.

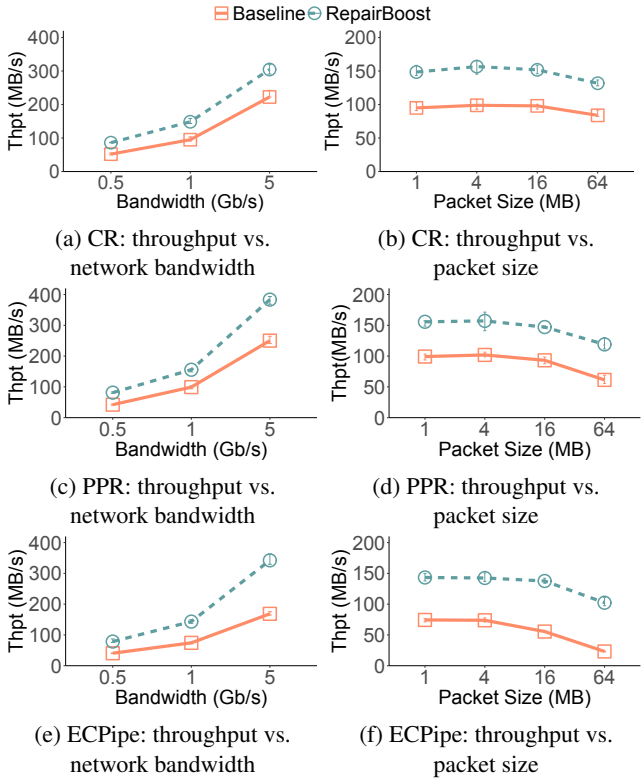
Table 1 gives the load balancing degrees of both the random selection and the LRU-based selection. We see that the random selection generates more balanced repair traffic in most cases. The reason is that the LRU-based selection prioritizes the nodes that have the smallest timestamps and IDs, and the repair traffic becomes progressively imbalanced when more chunks are repaired. Thus, we choose the random selection as the baseline for comparison.

Default configurations: Unless otherwise specified, we select the following default configurations throughout the evaluation. We set the chunk size to 64 MB (the default value in Hadoop HDFS) and the packet size to 1 MB. We mainly consider the following erasure codes: RS$(6,3)$ [6, 43], LRC$(6,2,2)$ [21], and Butterfly(4,2) [44]. We repair 100 chunks in each test and repeat it for five runs.

5.2 Experiments on Sensitivity

We first study the effectiveness of the parameters configured in RepairBoost on the actual repair performance.

[4] We use RC(k,m,d) to represent the regenerating code that encodes k data chunks into $k+m$ coded chunks, where a data chunk can be repaired by retrieving data from any $d \geq k$ surviving coded chunks [13].



(a) CR: throughput vs. network bandwidth

(b) CR: throughput vs. packet size

(c) PPR: throughput vs. network bandwidth

(d) PPR: throughput vs. packet size

(e) ECPipe: throughput vs. network bandwidth

(f) ECPipe: throughput vs. packet size

Figure 9: Experiment 1 (Impact of network bandwidth and packet size).

Experiment 1 (Impact of network bandwidth and packet size): We first assess the impact of network bandwidth on the repair throughput when coupling RepairBoost with existing repair algorithms. We vary the network bandwidth from 0.5 Gb/s (i.e., the network bandwidth dominates the repair) to 5 Gb/s (i.e., the disk bandwidth dominates the repair), and evaluate the repair throughput in different repair scenarios.

Figures 9(a), 9(c), and 9(e) show that the repair throughput generally increases with the available network bandwidth, as more data can be transmitted per time unit. Overall, RepairBoost can improve the repair throughput by 72.3% on average for different repair algorithms when compared to the baseline. In addition, we identify that RepairBoost is more advantageous in the network-bandwidth-dominated scenario, as RepairBoost balances the repair traffic and arranges the data transmission for maximizing the utilization of network bandwidth in repair. Specifically, the improvement of RepairBoost on the repair throughput increases from 53.0% (when the network bandwidth is 5 Gb/s) to 96.4% (when the network bandwidth is 0.5 Gb/s). It is worth noting that RepairBoost mainly considers the application scenario where the data transfers across the network dominate the repair performance, which conforms to the assumptions made by many previous studies [19, 40, 56, 58]. Even in the scenario with high-performance network systems (e.g., InfiniBand [45]) for fast memories (e.g., Intel Optane DIMM [61]), RepairBoost can still maintain its performance gain, since the network bandwidth is likely to be more stringent than the aggregated memory bandwidth (e.g., 44.8 Gb/s of the network bandwidth in RDMA versus 84.8Gb/s of the write bandwidth in six Optane DIMMs [26]).

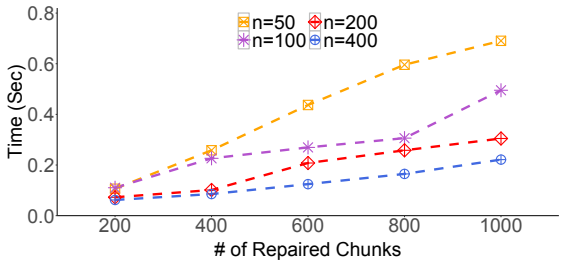


Figure 10: Experiment 2 (Computation time).

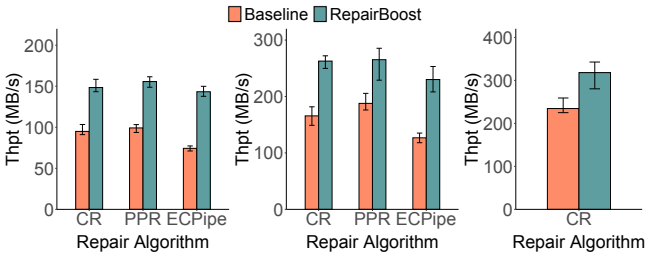
As RepairBoost employs multi-threading to pipeline the disk I/O and data transmission in repair, we also study the impact of packet size on the repair performance, where the packet size is changed from 1 MB (i.e., 64 packets per chunk) to 64 MB (i.e., one packet per chunk).

Figures 9(b), 9(d), and 9(f) indicate that the repair throughput decreases when the packet size increases, since the available disk and network bandwidth can be more extensively utilized in repair when a chunk comprises more packets. The multi-threading feature has no effect when the packet size reaches 64 MB (i.e., the chunk size), thereby leading to the lowest repair throughput. Overall, RepairBoost improves the repair throughput by 97.1% compared to the baseline under different packet sizes.

Experiment 2 (Computation time): We allocate one instance on Amazon EC2 with the same configurations as described in §5.1. We measure the computation time needed by RepairBoost to generate the repair solutions under different numbers of nodes (denoted by n) and repaired chunks.

Figure 10 shows the results. We make three observations. First, when the number of nodes is fixed, the computation time of RepairBoost gradually increases with the number of chunks being repaired, as RepairBoost has to process more RDAGs in the repair traffic balancing and transmission scheduling. Second, when the number of repaired chunks is fixed, the computation time drops when more nodes can participate in the repair, as RepairBoost can dispatch more chunks at a time. Third, the computation time needed by RepairBoost is always less than 0.9 seconds, thereby demonstrating that it is qualified to be deployed in the online repair scenario.

As the coordinator is only in charge of solving for the repair solutions and interacting with agents (§4), this experiment can evaluate the scalability of RepairBoost to some extent. When RepairBoost is deployed in a large-scale system (e.g., with thousands of nodes), we offer two options to further reduce the computation time. First, we can break a full-node repair into several subprocesses and iteratively repair a small number of chunks for each subprocess. Second, we can compute the repair solutions in advance and perform the repair based on the pre-computed results when a failure happens.



(a) RS(6,3) (b) LRC(6,2,2) (c) Butterfly(4,2)

Figure 11: Experiment 3 (Generality).

5.3 Experiments on RepairBoost Property

We also analyze the properties of RepairBoost, in terms of generality and technique breakdown.

Experiment 3 (Generality): We validate the generality of RepairBoost using a variety of representative erasure codes. Given an erasure code, we measure the repair throughput when RepairBoost is coupled with different repair algorithms. Note that we only consider CR for Butterfly$(4,2)$ as it repairs a chunk with multiple sub-chunks coming from different surviving chunks.

Figure 11 shows that all the repair algorithms assisted by RepairBoost can boost the repair for different erasure codes, thereby demonstrating the generality of RepairBoost. Overall, RepairBoost can improve the repair throughput by 60.4% on average for different erasure codes when compared to the baseline. RS$(6,3)$ has the lowest repair throughput (Figure 11(a)), as most of the time it needs to retrieve the most chunks for repair. While LRC$(6,2,2)$ comprises the same number of data chunks (i.e., k) as RS$(6,3)$ within a stripe, it achieves a higher repair throughput (Figure 11(b)), as it only retrieves three chunks to accomplish most of a single-chunk repair. Butterfly$(4,2)$ reaches the highest repair throughput (Figure 11(c)), as it needs to fetch only half of the remaining data (i.e., $\frac{k+1}{2}$ surviving chunks on average per failed chunk).

We also identify that ECPipe reaches the lowest repair throughput for both RS$(6,3)$ and LRC$(6,2,2)$. The root cause is that an RDAG under ECPipe only has one chunk to be transmitted at a time, hence limiting the repair parallelism.

Experiment 4 (Breakdown analysis): We decompose RepairBoost to demonstrate the effectiveness of each designed technique. For simplicity of presentation, we abbreviate the techniques of RepairBoost as follows: (i) *repair traffic balancing* (RTB), which balances the upload and download repair traffic (§3.2) without performing transmission scheduling, and (ii) *transmission scheduling* (TS), which simply schedules data transmission (§3.3) without considering repair traffic balancing.

Figure 12 shows the repair throughput of the baseline, RTB, TS, and RepairBoost. We make two observations. First, the effectiveness of RTB and TS varies across different repair algorithms. For example, TS outperforms RTB for PPR and ECPipe, but reaches a lower repair throughput for CR. Second, RepairBoost always reaches the highest repair throughput, indicating that the two techniques in RepairBoost are complementary without comprising the effectiveness of each other. Specifically, RepairBoost achieves 45.7% and 19.8% higher repair throughput than RTB and TS, respectively.

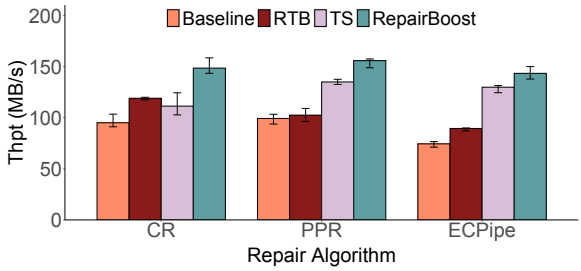


Figure 12: Experiment 4 (Breakdown analysis).

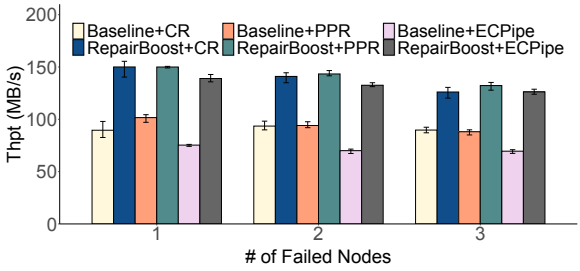


Figure 13: Experiment 5 (Performance on multi-node repair).

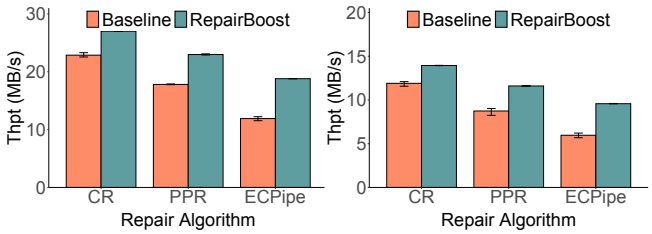
5.4 Experiments on Practicality

We further assess the practicality of RepairBoost by measuring its performance when tackling multi-failure repair and the single-node failure in heterogeneous environments.

Experiment 5 (Performance on multi-node repair): We extend RepairBoost and study its performance when repairing multiple failed nodes. We erase the data stored in a number of nodes to mimic multiple node failures. We then schedule the RDAGs of the lost chunks for repair. As RS$(6,3)$ can tolerate at most three node failures, this experiment measures the repair throughput when the number of failed nodes increases from one to three.

Figure 13 indicates that RepairBoost also speeds up the repair for multiple failures. Specifically, RepairBoost can improve the repair throughput by 39.5% (when tackling a single node failure) and by 35.7% (when tackling triple node failures). The repair throughput gained by RepairBoost drops slightly when more nodes fail, as fewer surviving nodes can be selected for RepairBoost to balance repair traffic and schedule transmission.

Experiment 6 (Performance with bandwidth heterogeneity): We finally assess the performance of RepairBoost in the presence of bandwidth heterogeneity. We organize the 16 instances with agents into four clusters, where each cluster comprises four instances and the intra-cluster bandwidth is 1 Gb/s. We then use the Linux bandwidth control tool `tc` to throttle the inter-cluster bandwidth for producing the bandwidth diver-



(a) Inter-cluster bandwidth: 0.1 Gb/s

(b) Inter-cluster bandwidth: 0.05 Gb/s

Figure 14: Experiment 6 (Performance with bandwidth heterogeneity).

sity phenomenon in data centers [10,15]. Specifically, we vary the inter-cluster bandwidth from 0.05 Gb/s to 0.1 Gb/s, such that the over-subscription ratio (i.e., calculated by dividing the intra-cluster bandwidth by the inter-cluster bandwidth) ranges from 10 to 20, as shown in prior studies [10,15,55].

Figure 14 shows that RepairBoost retains its effectiveness on accelerating the full-node repair in heterogeneous environments, especially when the inter-cluster bandwidth is more scarce. Specifically, the improvement of repair throughput gained by RepairBoost is 35.0% when the inter-cluster bandwidth is 0.1 Gb/s (Figure 14(a)), and increases to 36.8% when the inter-cluster bandwidth reduces to 0.05 Gb/s (Figure 14(b)).

6 Related Work

We review existing studies to accelerate full-node repair from the following aspects: repair-efficient codes, repair algorithms, and proactive repair with erasure coding.

Repair-efficient codes: LRCs [21,52] and Rotated-RS codes [27] reduce the repair traffic via loosening the storage efficiency requirement (i.e., increasing a handful of storage overhead). Regenerating codes [13] alleviate the repair traffic by requiring the surviving nodes to send a linear combination of the locally stored data or requiring more surviving nodes to participate in the repair (e.g., product-matrix MSR codes [49]). Butterfly code [44] and Clay code [58] can directly send the locally stored data for repair. Hitchhiker [48] makes the chunks across stripes dependent in the code construction for reducing the repair traffic. Different from the concrete constructions of erasure codes, RepairBoost is a scheduling approach that can assist the full-node repair for a variety of erasure codes.

Repair algorithms: Degraded-first scheduling [31] proposes to launch the degraded read tasks with a higher priority to leverage the unused network bandwidth. PUSH [22] pipelines the transmission of the requested chunks to alleviate the network congestion in repair. PPR [40] partitions an entire repair solution into small sub-stages and executes them in parallel. ECPipe [32] further partitions a chunk into smaller slices and pipelines the transmission of the slices, such that the complexity of the repair time approaches to O(1) in homogeneous environment. In view of the bandwidth diversity in hierarchical data centers, CAR [56] and ClusterSR [55] reduce and balance the inter-cluster repair traffic, which is considered more scarce than intra-cluster repair traffic. ECWide [18] addresses the repair of wide stripes with ultra-low redundancy in hierarchical data centers. Most of these studies pay special attention to the single-chunk repair, while RepairBoost can help different repair algorithms accelerate the full-node repair.

Proactive repair with erasure coding: Existing mature machine-learning-based failure prediction often take input the SMART attributes [11,17,30,38,64] combined with additional system events [60] and performance metrics [37], and exhibit high prediction accuracy (e.g., at least 95% [11,30,39,64]) with low false alarm rate (also called false positive rate, e.g., up to 2.5% [38], 0.2-0.4% [63]). FastPR [54] couples migration and erasure-coding-based repair to accelerate the proactive repair. Hu et al. [20] suggest proactively launching degraded reads to bypassing the hotspots, so as to reduce the tail latency in read operations. As an orthogonal study, RepairBoost is also applicable to the proactive repair that employs erasure coding for data recovery.

7 Conclusion

Erasure coding is a storage-efficient means to assure data reliability, yet it is prone to magnify the repair traffic. We present RepairBoost, a scheduling framework that boosts the full-node repair for various erasure codes and repair algorithms. RepairBoost employs a graph abstraction, called an RDAG, to characterize the single-chunk repair solution. It then carefully assigns the repair tasks of the RDAGs to the nodes for balancing the upload and download repair traffic. RepairBoost further schedules the transmission of chunks to saturate the unoccupied bandwidths. Extensive experiments on Amazon EC2 demonstrate the generality, flexibility, and effectiveness of RepairBoost.

Acknowledgments

We thank our shepherd, Mike Mesnier, and the anonymous reviewers for the comments on our camera-ready preparations. This work is supported by Natural Science Foundation of China (62072381, 61832011), CCF-Tencent Open Fund WeBank Special Fund, Xiamen Youth Innovation Fund (3502Z20206052), Zhejiang Lab (2021KF0AB01), the Natural Science Foundation of Fujian Province of China (2020J01002), and Research Grants Council of Hong Kong (AoE/P-404/18).

References

[1] Intel(R) Intelligent Storage Acceleration Library. https://github.com/intel/isa-l.

[2] Jerasure: Erasure Coding Library. https://jerasure.org/.

[3] Redis. https://redis.io/.

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

[5] Yahoo Cloud Object Store - Object Storage at Exabyte Scale. https://yahooeng.tumblr.com/post/116391291701/yahoo-cloud-object-store-object-storage-at, 2015.

[6] Apache Hadoop 3.1.4. https://hadoop.apache.org/docs/r3.1.4/, 2020.

[7] HDFS Erasure Coding. https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSErasureCoding.html, 2020.

[8] Amazon EC2. https://aws.amazon.com/ec2/, 2021.

[9] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris. Scarlett: Coping with Skewed Content Popularity in Mapreduce Clusters. In *Proc. of ACM EuroSys*, 2011.

[10] T. Benson, A. Akella, and D. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proc. of ACM IMC*, 2010.

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

[12] M. Chowdhury, Z. Liu, A. Ghodsi, and I. Stoica. HUG: Multi-Resource Fairness for Correlated and Elastic Demands. In *Proc. of USENIX NSDI*, 2016.

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

[14] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, 2003.

[15] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proc. of ACM SIGCOMM*, 2009.

[16] G. Hamerly and C. Elkan. Bayesian Approaches to Failure Prediction for Disk Drives. In *Proc. of ICML*, 2001.

[17] S. Han, P. P. C. Lee, Z. Shen, C. He, Y. Liu, and T. Huang. Toward Adaptive Disk Failure Prediction via Stream Mining. In *Proc. of IEEE ICDCS*, 2020.

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

[19] Y. Hu, X. Li, M. Zhang, P. P. C. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice. *ACM Transactions on Storage*, 13(4):33, 2017.

[20] Y. Hu, Y. Wang, B. Liu, D. Niu, and C. Huang. Latency Reduction and Load Balancing in Coded Storage Systems. In *Proc. of ACM SoCC*, 2017.

[21] C. Huang, H. Simitci, Y. Xu, et al. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.

[22] J. Huang, X. Liang, X. Qin, Q. Cao, and C. Xie. Push: A Pipelined Reconstruction I/O for Erasure-Coded Storage Clusters. *IEEE Transactions on Parallel and Distributed Systems*, 26(2):516–526, 2014.

[23] Q. Huang, K. Birman, R. Van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *Proc. of ACM SOSP*, 2013.

[24] A. Itai, Y. Perl, and Y. Shiloach. The Complexity of Finding Maximum Disjoint Paths with Length Constraints. *Networks*, 12(3):277–286, 1982.

[25] S. Kadekodi, K. Rashmi, and G. R. Ganger. Cluster Storage Systems Gotta Have HeART: Improving Storage Efficiency by Exploiting Disk-Reliability Heterogeneity. In *Proc. of USENIX FAST*, 2019.

[26] A. Kalia, D. Andersen, and M. Kaminsky. Challenges and Solutions for Fast Remote Persistent Memory Access. In *Proc. of ACM SoCC*, 2020.

[27] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, 2012.

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

[29] H. Li, Y. Zhang, D. Li, Z. Zhang, S. Liu, P. Huang, Z. Qin, K. Chen, and Y. Xiong. URSA: Hybrid Block Storage for Cloud-Scale Virtual Disks. In *Proc. of EuroSys*, 2019.

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

[31] R. Li, P. P. Lee, and Y. Hu. Degraded-First Scheduling for Mapreduce in Erasure-Coded Storage Clusters. In *Proc. of IEEE/IFIP DSN*, 2014.

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

[33] X. Li, R. Li, P. P. C. Lee, and Y. Hu. OpenEC: Toward Unified and Configurable Erasure Coding Management

in Distributed Storage Systems. In *Proc. of USENIX FAST*, 2019.

[34] A. Liaw, M. Wiener, et al. Classification and Regression by randomForest. *R news*, 2(3):18–22, 2002.

[35] Q. Lin, K. Hsieh, Y. Dang, H. Zhang, K. Sui, Y. Xu, J.-G. Lou, C. Li, Y. Wu, R. Yao, et al. Predicting Node Failure in Cloud Service Systems. In *Proc. of ESEC/FSE*, 2018.

[36] H. Liu, M. Mukerjee, C. Li, et al. Scheduling Techniques for Hybrid Circuit/Packet Networks. In *Proc. of ACM CoNEXT*, 2015.

[37] S. Lu, B. Luo, T. Patel, Y. Yao, D. Tiwari, and W. Shi. Making Disk Failure Predictions SMARTer! In *Proc. of USENIX FAST*, 2020.

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

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

[40] S. Mitra, R. Panta, M. Ra, and S. Bagchi. Partial-Parallel-Repair (PPR): A Distributed Technique for Repairing Erasure Coded Storage. In *Proc. of ACM EuroSys*, 2016.

[41] S. Muralidhar, W. Lloyd, S. Roy, et al. f4: Facebook's Warm Blob Storage System. In *Proc. of USENIX OSDI*, 2014.

[42] J. F. Murray, G. F. Hughes, and K. Kreutz-Delgado. Machine Learning Methods for Predicting Failures in Hard Drives: A Multiple-Instance Application. *Journal of Machine Learning Research*, 2005.

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

[44] L. Pamies-Juarez, F. Blagojevic, R. Mateescu, C. Gyuot, E. E. Gad, and Z. Bandic. Opening the Chrysalis: On the Real Repair Performance of MSR Codes. In *Proc. of USENIX FAST*, 2016.

[45] G. F. Pfister. An Introduction to the Infiniband Architecture. *High performance mass storage and parallel I/O*, 42(617-632):102, 2001.

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

[47] J. S. Plank and C. Huang. Tutorial: Erasure Coding for Storage Applications. Slides presented at USENIX FAST 2013, Feb 2013.

[48] K. Rashmi, N. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Hitchhiker's Guide to Fast and Efficient Data Reconstruction in Erasure-Coded Data Centers. In *Proc. of ACM SIGCOMM*, 2015.

[49] K. V. Rashmi, N. Shah, and P. V. Kumar. Optimal Exact-Regenerating Codes for Distributed Storage at The MSR and MBR Points via A Product-Matrix Construction. *IEEE Transactions on Information Theory*, 57(8):5227–5239, 2011.

[50] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook-Warehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.

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

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

[53] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *IEEE Transactions on Information Theory*, 58(4):2134–2158, 2011.

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

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

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

[57] R. E. Tarjan. *Data Structures and Network Algorithms*. SIAM, 1983.

[58] M. Vajha, V. Ramkumar, B. Puranik, G. Kini, E. Lobo, B. Sasidharan, P. V. Kumar, A. Barg, M. Ye, S. Narayanamurthy, et al. Clay Codes: Moulding MDS Codes to Yield an MSR Code. In *Proc. of USENIX FAST*, 2018.

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

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

[61] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and

Use of Scalable Persistent Memory. In *Proc. of USENIX FAST*, 2020.

[62] H. Zhang, M. Dong, and H. Chen. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *Proc. of USENIX FAST*, 2016.

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

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