Boosting Degraded Reads in Heterogeneous Erasure-Coded Storage Systems

Yunfeng Zhu, Jian Lin, Patrick P. C. Lee, and Yinlong Xu

Abstract—Distributed storage systems provide large-scale data storage services, yet they are confronted with frequent node failures. To ensure data availability, a storage system often introduces data redundancy via replication or erasure coding. As erasure coding incurs significantly less redundancy overhead than replication under the same fault tolerance, it has been increasingly adopted in large-scale storage systems. In erasure-coded storage systems, degraded reads to temporarily unavailable data are very common, and hence boosting the performance of degraded reads becomes important. One challenge is that storage nodes tend to be heterogeneous with different storage capacities and I/O bandwidths. To this end, we propose FastDR, a system that addresses node heterogeneity and exploits I/O parallelism, so as to boost the performance of degraded reads to temporarily unavailable data. FastDR incorporates a greedy algorithm that seeks to reduce the data transfer cost of reading surviving data for degraded reads, while allowing the search of the efficient degraded read solution to be completed in a timely manner. We implement a FastDR prototype, and conduct extensive evaluation through simulation studies as well as testbed experiments on a Hadoop cluster with 10 storage nodes. We demonstrate that our FastDR achieves efficient degraded reads compared to existing approaches.

Keywords—Erasure-coded storage system, degraded reads, node heterogeneity, I/O parallelism

1 Introduction

Distributed storage systems, such as GFS [11] and Azure [5], have been widely adopted in enterprises to provide large-scale storage services. However, component failures are frequent and diverse in large-scale storage systems [10], [11], [17], [29]. To ensure data availability, storage systems usually stripe data redundancy across multiple storage nodes (or servers). Replication is traditionally used to provide data redundancy [5], [11], yet it introduces high storage overhead and becomes a scalability bottleneck. On the other hand, *erasure coding* provides space-optimal data redundancy while achieving the same fault tolerance as replication [35]. It operates by encoding data into multiple fragments, such that any subset of fragments can sufficiently reconstruct the original data. Erasure coding has been widely deployed and evaluated in large-scale storage systems by both commercial and academic communities (e.g., [1], [3], [17], [19], [20], [24], [28], [29]).

Practical storage systems may experience two types of node failures: *permanent* and *temporary*. A node is permanently failed if it loses all its stored data. To preserve the required redundancy level and maintain data availability, a storage system performs *failure recovery*, which reconstructs all the lost data in another new node. Permanent failure recovery has always been important as shown in previous studies [16], [19], [33], [37], [41], [42]. On the other hand, a node is temporarily failed if its stored data is not lost but is only temporarily unavailable for direct accesses. To access the unavailable data, a storage system performs a *degraded read*, which retrieves data from surviving nodes and reconstructs the unavailable data. It is expected that degraded reads are slower than normal reads. Field measurements show that temporary failures contribute to the majority of component failures in large-scale storage systems [10]. Thus, degraded reads are more frequently performed than failure recovery operations, and their performance optimizations are critical.

One challenge of optimizing degraded reads, as opposed to failure recovery, is that a read request often works under a more stringent latency constraint than a recovery operation. Furthermore, degraded read optimizations must take into account the underlying configuration of the storage system. Due to system upgrades and scaling, distributed storage systems are typically composed of nodes with heterogeneous storage capacities and I/O speeds [21]–[23], [40]. Also, files are usually distributed over multiple storage nodes and accessed in parallel. Thus, the degraded read performance is inevitably bottlenecked by the poorly performed surviving nodes. On condition that data blocks in the degraded read requests are correctly reconstructed, it is necessary for degraded reads to account for both node heterogeneity and I/O parallelism, so that the performance of degraded reads is boosted.

This paper studies how to boost degraded reads in large-scale heterogeneous erasure-coded storage systems. We first formulate an optimization problem of boosting degraded reads, in which we associate each storage node with a cost of reading per unit of data and

- *Y. Zhu and Y. Xu are with AnHui Province Key Laboratory of High Performance Computing, School of Computer Science and Technology, University of Science & Technology of China (emails: zyfl@mail.ustc.edu.cn, ylxu@ustc.edu.cn)*
- *J. Lin and P. P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong (emails: linjiansuper@gmail.com, pclee@cse.cuhk.edu.hk)*

propose a model that minimizes the parallel degraded read time. We then propose an *enumerated greedy (EG)* algorithm to quickly search for an efficient solution for degraded reads. We emphasize that the EG algorithm addresses both node heterogeneity and I/O parallelism. Our simulations show that the EG algorithm significantly reduces the degraded read time compared to the baseline approach, while only introducing small computational overhead.

To validate the feasibility of deploying the EG algorithm in practical storage systems, we built a prototype system called *FastDR* on HDFS-RAID [14], an erasure coding extension to the Hadoop Distributed File System (HDFS) [30]. FastDR enhances the performance for degraded reads in heterogeneous erasure-coded storage systems through two major components: (1) using disk-oriented reconstruction (DOR) [15] to optimize the degraded read work flow and parallelize the read operation, and (2) using the EG algorithm to determine which blocks to be downloaded from surviving nodes, so as to efficiently fulfill degraded read requests in heterogeneous erasure-coded storage systems. We deploy FastDR in a HDFS cluster testbed composed of 10 heterogeneous storage nodes. Through extensive testbed experiments, we validate the improvement of FastDR over the baseline approach. We also show that our FastDR improves the performance of MapReduce [7] in HDFS in the presence of failures.

The remainder of the paper proceeds as follows. Section 2 elaborates the requirements to be considered for boosting degraded reads. Section 3 formulates the problem of degraded reads and proposes an enumerated greedy algorithm for degraded reads. Section 4 presents the design details of FastDR. Section 5 presents results of simulations and testbed experiments of FastDR. Section 6 reviews the related work, and finally, Section 7 concludes the paper.

2 Design Requirements

Reads on transient failed nodes are *degraded*, as the unavailable data needs to be reconstructed from other surviving nodes. Our goal is to enable low-latency degraded reads in practical erasure-coded storage systems in the presence of transient node failures, which are commonly found in real-life storage systems [10]. Specifically, we seek to address the following requirements when we design efficient degraded read solutions.

Range accesses: The key objective for permanent failure recovery is to read data from surviving nodes to reconstruct the lost data on a new replacement node. However, degraded reads are different in that a degraded read request typically accesses a range of data units that span not only the unavailable data units on the temporarily failed nodes, but also the available data units on the surviving nodes. To realize a degraded read, it is necessary to read the available data units and extra data units to reconstruct the unavailable data units from the surviving nodes.

Node heterogeneity: Storage systems are often upgraded over time, and thus storage nodes may be composed of heterogeneous hardware resources, such as disks and network interfaces with different bandwidths [40]. On the other hand, the available resources of each storage node may vary from time to time due to dynamic load conditions. Since degraded reads usually use the available data from multiple surviving nodes to reconstruct the unavailable data, the resulting degraded read performance may be bottlenecked by the poorly performed surviving nodes.

Parallel accesses: To further boost the degraded read performance, the degraded read solution should leverage I/O parallelism, which has become an essential property of modern storage systems. Contiguous blocks are striped across different nodes so that a sequential read request can be parallelized. Previous studies (e.g., [16], [33]) also exploit parallel I/Os in failure recovery. A challenge is to extend the parallelism in degraded reads subject to range accesses and node heterogeneity.

Online decision: The degraded read performance is determined by the read size and node resources, both of which are variable factors depending on the current load conditions. Thus, the degraded read solution must be determined online for each read request based on its read size and the available node resources. In particular, it is infeasible to enumerate on-the-fly the degraded read solutions for all possible failures as in [19], but instead we need to find an efficient degraded read solution in a timely manner.

Applicability to general erasure codes: Authors of [17], [19] propose new erasure codes which achieve efficient degraded read performance. However, a storage system has typically deployed a specific erasure code for its stored data, and re-encoding a huge amount of stored data with a new code will introduce significant overheads. Also, different types of data are to be differently encoded based on the fault tolerance and access requirements [1]. Thus, instead of constructing new codes, we aim to develop efficient degraded read strategies that are applicable to a general class of existing erasure codes.

3 Enumerated Greedy Algorithm

In this section, we propose the *enumerated greedy (EG)* algorithm for degraded reads in erasure-coded storage systems, so as to efficiently reconstruct lost data in heterogeneous settings. We focus on a special family of erasure codes called *XOR-based erasure codes*, in which both encoding and decoding involve XOR operations only. We first formulate the problem of degraded reads for XOR-based erasure codes, and then propose an optimization model for degraded reads that accounts for node heterogeneity and I/O parallelism. We further present our EG algorithm that can return an efficient degraded read solution within a reasonable search time. Table 1 summarizes the major notation used throughout this paper.

TABLE 1
Major notation used throughout this paper.

Symbol	Description
n	number of storage nodes
k	number of data nodes
m	number of parity nodes
w	number of blocks per strip
l	number of blocks in a degraded request
f	number of the failed data nodes
d	number of surviving nodes connected for realizing degraded read requests ($k \leq d \leq n - f$)
c_i	time cost for reading one block from *Node* i ($0 \leq i \leq n-1$)
$x_{u,v}$	indicator variable of reading the v-th block in a stripe to reconstruct the u-th block in the read request ($0 \leq u \leq l$, $0 \leq v \leq nw-1$)

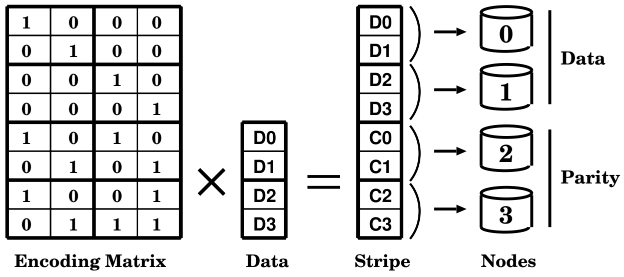
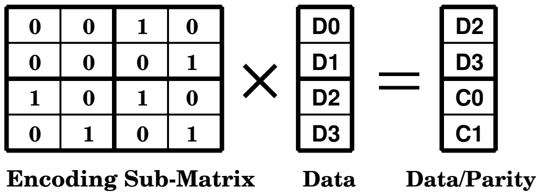


Fig. 1. Example of how a stripe is encoded for $(k, m, w) = (2, 2, 2)$.

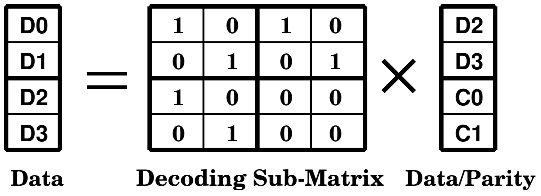
3.1 Problem Formulation

We consider an erasure-coded storage system is composed of n storage nodes with k data nodes and m parity nodes (i.e., $n = k + m$). We assume that the system can tolerate any m out of n node failures while maintaining storage optimality, and we call this property the *Maximum Distance Separable (MDS) property*. We focus on the *systematic* erasure codes, such that the data nodes contain the original data. The storage system is partitioned into *stripes*. Each stripe contains nw *blocks*, where w data (parity) blocks are stored in each data (parity) node. Suppose that the j-th block in *Node* i within a stripe is labeled as the v-th block in the stripe, where $v = w \times i + j$, $0 \leq i \leq n-1$ and $0 \leq j \leq w-1$. In other words, we have $0 \leq v \leq nw - 1$. Each parity block is encoded from the data blocks of the same stripe, by multiplying a $wn \times wk$ *encoding matrix* with a column vector of wk data blocks. We denote the above mentioned erasure code as (k, m, w). Figure 1 shows an example of a (2,2,2) erasure code based on Cauchy Reed-Solomon (CRS) codes [4]. Each stripe is independently encoded, so our analysis focuses on a single stripe [27], [32]. Note that the identities of data and parity nodes are usually rotated across stripes in the implementation of storage systems for load balancing.

Since each read request is issued to data nodes, we assume that failures appear in some of the k data nodes. Without loss of generality, let nodes $0, 1, \cdots, k-1$ be data nodes, and nodes $k, k+1, \cdots, n-1$ be parity nodes.



(a) Extract the encoding sub-matrix for Node 1 and Node 2 from the encoding matrix



(b) Compute the corresponding decoding sub-matrix for Node 1 and Node 2

Fig. 2. An example of encoding sub-matrix and decoding sub-matrix.

Suppose now that a read request accesses a collection of data blocks on some data nodes, some of which are failed. Let f be the number of failed nodes. We require that $f \leq m$, so that the failures are tolerable. Then the read request becomes *degraded*, as it needs to read the data blocks and parity blocks from other surviving nodes in order to reconstruct the lost data blocks.

Any erasure code can be defined by an encoding matrix [27], [32], from which we give the following definitions.

Definition *1:* An *encoding sub-matrix* is a square matrix extracted from the encoding matrix, and indicates exactly how data/parity blocks from the selected k out of n nodes are encoded from the original kw data blocks. □

In fact, any k out of n storage nodes correspond to exactly one encoding sub-matrix. Take CRS codes in Figure 1 as an example. Figure 2(a) shows the encoding sub-matrix extracted for *Node 1* and *Node 2*, and indicates how data/parity blocks D_2, D_3, C_0, and C_1 are encoded from blocks D_0, D_1, D_2, and D_3.

Definition *2:* A *decoding sub-matrix* is the inverse matrix computed from an encoding sub-matrix, and indicates how data blocks can be decoded by data/parity blocks from the selected k storage nodes. □

Due to the MDS property, each encoding sub-matrix is invertible. Through computing its inverse matrix, we can get the corresponding decoding sub-matrix. Figure 2(b) shows the decoding sub-matrix of the encoding sub-matrix in Figure 2(a), and indicates how data blocks D_0, D_1, D_2, and D_3 can be decoded from data/parity blocks D_2, D_3, C_0, and C_1.

Definition *3:* Given a decoding sub-matrix, each data block can be represented by the XOR-sum of a set of data and parity blocks, which is called a *combined degraded read equation (CDRE)*. □

In fact, each row in a decoding sub-matrix corresponds to a CDRE. For example, from the decoding sub-matrix shown in Figure 2(b), we get four CDREs:

$$\begin{aligned} D_0 &= D_2 \oplus C_0, \\ D_1 &= D_3 \oplus C_1, \\ D_2 &= D_2, \\ D_3 &= D_3. \end{aligned} \tag{1}$$

When some nodes fail, the decoding sub-matrices should be generated from surviving storage nodes. For example, when *Node* 0 fails in Figure 1, *Node 1* and *Node 2* can be used to generate a decoding sub-matrix. The above four CDREs are hence adopted for realizing a degraded read request. Suppose that we read data blocks D_0 (a lost block) and D_2 (a normal block). The degraded read request can be fulfilled with the following two CDREs:

$$\begin{aligned} D_0 &= D_2 \oplus C_0, \\ D_2 &= D_2. \end{aligned} \tag{2}$$

It will read two blocks $\{D_2, C_0\}$. In later discussion, we show how we use CDREs to search for an efficient degraded read solution.

3.2 Optimization Model

We now formalize the optimization problem for solving for a degraded read request in a heterogeneous environment. Let $x_{u,v}$ (where $0 \leq u \leq l-1$, $0 \leq v \leq nw-1$) be the indicator variable such that $x_{u,v} = 1$ if the v-th block in the stripe is read to reconstruct the u-th block in the request, and $x_{u,v} = 0$ otherwise. We also introduce a download distribution y_i to denote the number of blocks being read from *Node i* so as to realize a degraded read request, where $0 \leq i \leq n-1$. It can be computed as:

$$y_i = \sum_{j=0}^{w-1} \bigvee_{u=0}^{l-1} x_{u,w\times i+j}. \tag{3}$$

where $\bigvee$ denotes the OR operator, and $\bigvee_{u=0}^{l-1} x_{u,w\times i+j} = 1$ if $x_{u,w\times i+j} = 1$ for some u. The term $\bigvee_{u=0}^{l-1} x_{u,w\times i+j}$ determines whether the j-th block from *Node i* should be read for realizing a degraded read request.

We require that all $x_{u,v}$'s be chosen such that all data blocks in the degraded read request can be correctly reconstructed. In addition, we associate *Node i* ($0 \leq i \leq n-1$) with a time cost c_i of reading a block from *Node i*. Based on cost c_i ($0 \leq i \leq n-1$), we select a set $\mathcal{D}$ of d ($k \leq d \leq n-f$) surviving nodes to realize a degraded read request. That is, if the degraded read request reads data from the surviving node *Node i*, then we say $i \in \mathcal{D}$. Furthermore, we take into account I/O parallelism, and issue degraded reads to the surviving nodes in parallel. Let T be the time needed to download the necessary blocks from the slowest storage node to respond to the read request. With I/O parallelism, the value of T determines the degraded read performance. Our goal is to minimize T by carefully choosing $x_{u,v}$'s. We formulate the optimization problem with respect to $x_{u,v}$'s as follows:

$$\begin{aligned} \text{Minimize } \; T &= \max_{i\in\mathcal{D}}\{c_i y_i\} \\ &= \max_{i\in\mathcal{D}}\left\{c_i \sum_{j=0}^{w-1} \bigvee_{u=0}^{l-1} x_{u,w\times i+j}\right\}. \end{aligned} \tag{4}$$

Our optimization model assumes that the read time of each node is proportional to the number of blocks being read, and that the I/O seek time is ignored. Thus, we treat the sequential reads and random reads to the same number of blocks as having the same read time. We argue that if the block size is sufficiently large, the I/O seek overhead can be mitigated.

3.3 Solving the Model

To solve the heterogeneous model for degraded reads, we can adopt the *basic approach*, which builds on the MDS property. Its idea is to download the data and parity blocks from exactly k nodes, such that the downloaded blocks can be used to reconstruct the lost blocks correctly. In the case of single failure recovery, we assume that the basic approach reads the blocks from other $k-1$ surviving data nodes and the first parity node.

Recent studies [19], [37], [41] on failure recovery mainly aim to find the decoding equations that have maximum possible overlapping of blocks with each other. In other words, these approaches can be applied to degraded reads so as to minimize the number of blocks downloaded to fulfill the degraded requests. In fact, Khan *et al.* [19] show that the degraded read solution with the minimized download blocks performs roughly the same to the basic approach. Therefore, we only consider the basic approach in this paper as the baseline.

We illustrate via an example the basic approach. Consider the storage system in Figure 1. Let α (in units of Mb) be the size of one block. Figure 3 shows a heterogeneous environment for the storage system. Suppose that we make a read request to blocks D_0 and D_2 in the storage system. Then the basic approach finds its degraded read solution composed of two CDREs, as shown in Equation (2). It will read two blocks $\{D_2, C_0\}$, and hence has a degraded read time of $\max\{\alpha/38, \alpha/113\}$ = 0.0263α (in sec).

To find an optimal degraded read solution to Equation (4), one can use the *enumeration approach* [19], which selects an optimal collection of l *decoding equations*, each of which corresponds to a subset of blocks in the stripe such that a lost block can be recovered from the remaining blocks in the same subset. There are a total of $2^{nw}-1$ decoding equations, and the enumeration approach needs to check $\binom{2^{nw}-1}{l}$ combinations of l decoding equations. Traversing the entire search space is computationally expensive. Even for the simple example in the above paragraph, the search space is of size up to 32,385. Although search space can be pruned [19], the search time remains huge and violates our online decision requirement.

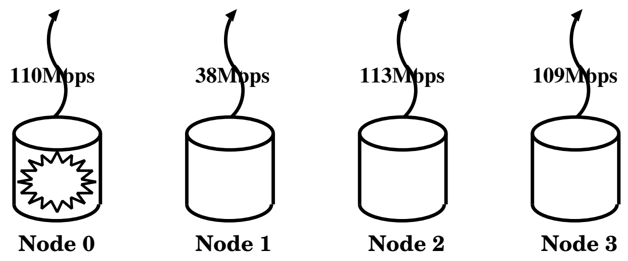
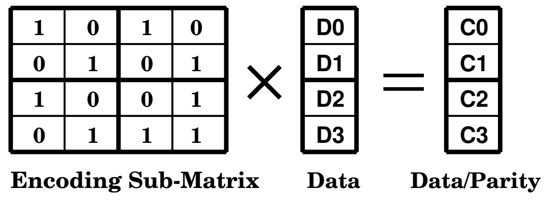
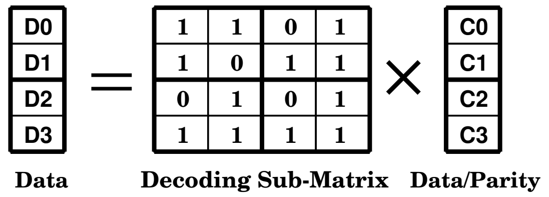


Fig. 3. Example of a single-node failure in heterogeneous storage environments.



(a) Extract the encoding sub-matrix for Node 2 and Node 3 from the encoding matrix



(b) Compute the corresponding decoding sub-matrix for Node 2 and Node 3

Fig. 4. An example of how to fulfill the degraded read request, bypassing surviving nodes with lower bandwidth.

We now consider an alternative approach that provides an efficient degraded read solution based on the CDREs. Our motivation is that in a heterogeneous storage environment, it is more preferred for a degraded read request to read blocks from surviving nodes with higher bandwidth. Suppose that a degraded read request reads blocks from the top d ($k \leq d \leq n - f$) surviving nodes with the highest bandwidth, where d is a configurable parameter. Our goal is to read the set of blocks from the d surviving nodes such that the degraded read time T is minimized.

We illustrate the idea via an example. We still consider the erasure code in Figure 1, and suppose that we read blocks D_0 and D_2. Obviously, *Node* 1 is a bottlenecked node that should be bypassed. Thus, we can set $d = 2$, and choose blocks from *Node* 2 and *Node* 3 to fulfill the degraded read request. Figure 4(a) shows the encoding sub-matrix, which indicates how blocks in *Node* 2 and *Node* 3 are encoded from data blocks according to the encoding matrix in Figure 1. Through computing its inverse matrix, we show in Figure 4(b) how data blocks D_0 and D_2 can be decoded from blocks in *Node* 2 and *Node* 3. Thus, the degraded read request can be fulfilled with a group of two CDREs as follows:

$$\begin{aligned} D_0 &= C_0 \oplus C_1 \oplus C3, \\ D_2 &= C_1 \oplus C3. \end{aligned} \tag{5}$$

It will read three blocks $\{C_0, C_1, C_3\}$. The degraded read time is $\max\{2\alpha/113, \alpha/109\} = 0.0177\alpha$ (in sec), which reduces the degraded read time of the basic approach by 32.70%.

Given a (k, m, w) erasure code (where $n = k + m$), suppose that storage system has f failed nodes, and we set $d \leq n - f$. From any k out of d surviving nodes, we can obtain a decoding sub-matrix due to the MDS property. Thus, there exist a total of $\binom{d}{k}$ CDREs for each data block in a stripe. To realize a degraded read request with l (where $0 \leq l \leq kw - 1$) data blocks, it is necessary to reconstruct l blocks with a group of l CDREs. The size of the solution space is $\binom{d}{k}^l$. Our goal is to find a degraded read solution that minimizes the total time of reading blocks from surviving nodes in a heterogeneous environment as described in Equation (4).

3.4 Enumerated Greedy (EG) Algorithm

As there are up to $\binom{d}{k}^l$ possible solutions to realize the degraded read request, it would be time-consuming to check all solutions to find the optimal solution. In this subsection, we propose an *enumerated greedy (EG)* algorithm, whose goal is to find an efficient degraded read solution (denoted by $\mathcal{R}$) that has near-optimal parallel degraded read time T, while being able to return the solution in a timely manner.

We define the notation for our EC algorithm. Consider a storage system that deploys an erasure code (k, m, w). Let $\mathcal{F}$ be the set of all failed nodes, and $\mathcal{C}$ be the vector that denotes the costs of all nodes and can be dynamically obtained based on the current system loads. Let $\mathcal{L}$ be the set of l data blocks to be read. Let $\mathcal{D}$ be the set of d surviving nodes from which we read the data and parity blocks to realize the degraded read request.

Functions: Our EG algorithm is built on six functions.

- BOTTLENECKNODES($\mathcal{C}$, $\mathcal{F}$, d): Given a cost vector $\mathcal{C}$ and a collection $\mathcal{F}$ of the failed nodes, the function returns a set $\mathcal{D}$ of d surviving nodes with the lowest costs.
- INVERSEMATRIX($\mathbb{E}, \mathcal{I}, \mathcal{D}$): Given an encoding matrix $\mathbb{E}$, a bitmap $\mathcal{I}$ which identifies the selected k out of d surviving nodes in $\mathcal{D}$, the function extracts the encoding sub-matrix for the k surviving nodes and returns the corresponding decoding sub-matrix.
- EXTRACTDRS($\mathbb{D}$, $\mathcal{L}$): Given a decoding sub-matrix $\mathbb{D}$ and a set $\mathcal{L}$ of data blocks in a degraded read request, the function returns a set of l CDREs for the data blocks in $\mathcal{L}$.
- NEXTBITMAP(d, $\mathcal{I}$): The function generates all options of bitmaps in lexicographic order, each containing k 1-bits and $d - k$ 0-bits to identify k out of d surviving nodes. It returns the next bitmap in order, or NULL if all bitmaps are enumerated.

- DEGRADEDREADTIME($X, \mathcal{C}$): The function computes the degraded read time for the CDRE X using Equation (4).
- REDUCEREADTIME($\mathcal{R}$, i, $\mathcal{C}$, X): The function substitutes the i-th CDRE in $\mathcal{R}$ with the CDRE X, and then gets a new degraded read solution $\mathcal{R}'$. Based on Equation (4), it then computes the parallel degraded read times of $\mathcal{R}$ and $\mathcal{R}'$. Finally, it returns the reduction of degraded read time of $\mathcal{R}'$ over $\mathcal{R}$ (a negative value means $\mathcal{R}$ has a smaller value of degraded read time).

Algorithm details: Figure 5 shows the EG algorithm. We first initialize the set $\mathcal{D}$ of d nodes with the lowest costs using the function BOTTLENECKNODES (Step 1). We also initialize the bitmap $\mathcal{I}$ with k 1-bits followed by $d-k$ 0-bits (Step 3).

Given k surviving nodes, in Steps 5-15, we calculate the minimal degraded read time for each block (*recorded in an array* $\mathcal{E}$) in the degraded read request $\mathcal{L}$. We traverse all possible choices of k surviving nodes, each of which will return a set of l CDREs for $\mathcal{L}$ using functions INVERSEMATRIX and EXTRACTDRS (Steps 6-7). We then compute the degraded read time for each CDRE using function DEGRADEDREADTIME (Step 9) and update the degraded read time if it is smaller than the current degraded read time (Steps 10-12). We repeat the above steps with other k surviving nodes via selecting the next bitmap $\mathcal{I}$ using the function NEXTBITMAP (Step 14).

We further re-traverse all possible choices of k surviving nodes, so as to find the efficient degraded read solution $\mathcal{R}$ for L (Steps 19-37). For each choice of k nodes, we extract l CDREs (Steps 20-21). We only consider the CDRE with the minimal degraded read time (Steps 25-33). For the first CDRE we consider, we use it to initialize the degraded read solution $\mathcal{R}$ (Steps 25-27). After that, we identify whether the CDRE with the minimal degraded read time can reduce the whole degraded read time of $\mathcal{R}$ using the function REDUCEREADTIME, and then update the degraded read solution $\mathcal{R}$ if it can do so (Steps 29-32). We repeat the process until all possible choices of k surviving nodes are traversed.

Algorithm complexity: The while-loop of Steps 5-15 takes $O(\binom{d}{k})$ time, and Steps 8-13 repeats the for-loop for l times. Also, the while-loop of Steps 19-37 takes $O(\binom{d}{k})$ time, and Steps 22-35 repeat the for-loop for l times. Therefore, the total time complexity of the EG algorithm is $O(\binom{d}{k}l) = O(\binom{d}{d-k}l)$, where $d-k$ is smaller than m (the number of tolerable failures). Through adjusting the parameter d, we can limit the search complexity of our EG algorithm. Our experimental results show that the EG algorithm can significantly improve degraded read performance within a reasonable time delay (see Section 5.1).

Algorithm: EG(l, d, $\mathcal{C}$, $\mathcal{F}$, $\mathcal{L}$)

```
Input:
  $l$: size of a degraded request
  $d$: configured number of surviving nodes
  $\mathcal{C}$: set of costs of storage nodes
  $\mathcal{F}$: set of failed nodes
  $\mathcal{L}$: set of blocks in a degraded read request
Output:
  $\mathcal{R}$: efficient solution for the degraded read request

1: Initialize $D$ = BOTTLENECKNODES($\mathcal{C}$, $\mathcal{F}$, $d$)
2: Initialize $\mathcal{E}$ with $l$ MAX_VALUE
3: Initialize $\mathcal{I}$ with $k$ 1-bits followed by $(d-k)$ 0-bits
4: /* Traverse all collections of k surviving nodes */
5: while $\mathcal{I} \neq$ NULL do
6:     $\mathbb{D}$ = INVERSEMATRIX($\mathbb{E}$, $\mathcal{I}$, $\mathcal{D}$)
7:     $\mathcal{X}$ = EXTRACTDRS($\mathbb{D}$, $\mathcal{L}$)
8:     for $0 \leq i < l$ do
9:         $T$ = DEGRADEDREADTIME($\mathcal{X}[i]$, $\mathcal{C}$)
10:        if $T < \mathcal{E}[i]$ then
11:            $\mathcal{E}[i] = T$
12:        end if
13:    end for
14:    $\mathcal{I}$ = NEXTBITMAP($d$, $\mathcal{I}$)
15: end while
16: $f$ = true
17: Initialize $\mathcal{I}$ with $k$ 1-bits followed by $(d-k)$ 0-bits
18: /* Re-traverse all collections of k surviving nodes */
19: while $\mathcal{I} \neq$ NULL do
20:    $\mathbb{D}$ = INVERSEMATRIX($\mathbb{E}$, $\mathcal{I}$, $\mathcal{D}$)
21:    $\mathcal{X}$ = EXTRACTDRS($\mathbb{D}$, $\mathcal{L}$)
22:    for $0 \leq i < l$ do
23:        $T$ = DEGRADEDREADTIME($\mathcal{X}[i]$, $\mathcal{C}$)
24:        if $T == \mathcal{E}[i]$ then
25:            if $f$ == true then
26:                $\mathcal{R}[i] = \mathcal{X}[i]$
27:                $f$ = false
28:            else
29:                $T'$ = REUDCEREADTIME($\mathcal{R}$, $i$, $\mathcal{C}$, $\mathcal{X}[i]$)
30:                if $T' > 0$ then
31:                    $\mathcal{R}[i] = \mathcal{X}[i]$
32:                end if
33:            end if
34:        end if
35:    end for
36:    $\mathcal{I}$ = NEXTBITMAP($d$, $\mathcal{I}$)
37: end while
38: Return $\mathcal{R}$
```

Fig. 5. The enumerated greedy (EG) algorithm.

4 FASTDR DESIGN

We evaluate the EG algorithm in a real network setting. We design and implement *FastDR*, a system that realizes the EG algorithm and is deployable on HDFS-RAID [14], an erasure-coded extension of Hadoop Distributed File System (HDFS) [30]. We first give an overview of HDFS-RAID. We then present how we design FastDR and integrate it into HDFS-RAID. Finally, we describe the implementation details of FastDR.

4.1 Overview of HDFS-RAID

HDFS-RAID [14] extends HDFS [30] with the feature of storing data using erasure coding, so as to reduce the storage overhead while preserving data availability. HDFS is a distributed file system designed for handling large-scale datasets with commodity hardware. It is composed of a single *NameNode* for metadata management and multiple *DataNodes* for data storage. DataNodes periodically report their status to the NameNode through heartbeat messages. An HDFS *client* interacts with NameNode and DataNodes to complete read/write operations. Specifically, for a read request, the client first obtains the locations of target data from the NameNode, and then downloads the data from the corresponding DataNodes. For a write request, the client first asks the NameNode for the DataNode where the data will be stored, and then directly sends the data to the target DataNode.

HDFS stores data in units of *blocks*. By default, HDFS achieves fault tolerance via replication, and stores three copies for each block. HDFS-RAID augments HDFS by grouping blocks into stripes and converting redundant replicas into erasure-coded blocks. HDFS-RAID runs on HDFS, and leverages HDFS for the read/write operations. In addition, it adds a *RaidNode* to support the conversion of replicas to erasure-coded data. HDFS-RAID also supports degraded reads, i.e., when a client wants to read an unavailable block, it downloads the necessary data and parity blocks from other surviving nodes and reconstructs the unavailable data blocks.

4.2 Design Components

Figure 6 shows the architecture of FastDR. FastDR relies on HDFS to handle write requests and HDFS-RAID to handle striping and recovery. During degraded reads, FastDR parses the current conditions of DataNodes and then determines the blocks to be downloaded from the surviving DataNodes. FastDR has three components: (i) *Cost Measurement Module*, which generates a cost vector between the client and DataNodes for degraded reads, (ii) *Failure Detector*, which determines whether the client needs to perform degraded reads, and (iii) *Degraded Read Handler*, which decides the blocks to read to fulfill the degraded read request.

The FastDR client handles read requests and can work in two different modes depending on the system status. For a read request, the FastDR client first passes it to the Failure Detector, which monitors the existence of any failed DataNode. If there is no failure detected, the FastDR client works in *normal mode*. The requested data is then directly read from corresponding DataNodes and passed to the client. If there is any related failure detected, the FastDR client will enter *degraded read mode*. The Degraded Read Component will handle this request and do any necessary reconstruction based on the cost vector generated by the Cost Measurement Module. In both normal and degraded read modes, the FastDR client downloads data from DataNodes with *node oriented parallelism*, whose idea is similar to disk-oriented reconstruction in RAID [16]. Specifically, it creates multiple threads that run in parallel. Each thread is associated with a DataNode, and downloads and decodes data from the associated DataNode.

In the following, we provide design and implementation details for the three main components of FastDR.

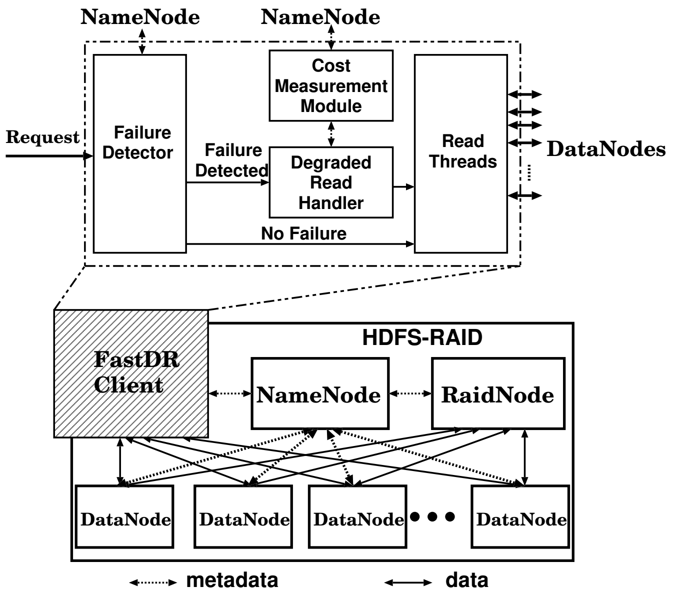


Fig. 6. The architecture of FastDR.

4.2.1 Cost Measurement

The Cost Measurement Module of FastDR aims to generate a cost vector between the FastDR client and DataNodes. It indicates the related cost for downloading data from DataNodes to client accurately.

The cost can be computed as the sum of two parts. The first part is the *static cost*, which is measured based on the static components such as hardware configurations. We assume that the static cost does not change from time to time. Thus, we measure this part before the system starts and hardcode the results as configuration parameters for bootstrapping the system. In our current implementation, we measure the static cost as the sum of the inverse of the speed of the network interface card and the inverse of the speed of raw disk read.

The second part is the *dynamic cost*, which is measured based on the current system workload and working mode. In our current implementation, we do not include the dynamic cost. We discuss two possible approaches to measure the dynamic cost. In the first approach, the NameNode periodically measures (e.g., via probing) the response time to each DataNode, and updates the dynamic cost via weighted average [6]. Another approach is to measure the cost based on popularity [2], in which the NameNode measures the access rate of each DataNode and associates a higher cost with the DataNode with a higher access rate. The rationale is that popular DataNodes form hotspots and have less effective bandwidth for data transfer. How to effectively measure the dynamic cost is posed as future work.

4.2.2 Failure Detection

In our current implementation and experiments, we simply hardcode the failure status in the Failure Detector, which directly triggers the degraded read mode. In practice, the Failure Detector is activated at the beginning of a read request. Recall that the NameNode receives periodic status reports from DataNodes (see Section 4.1). Thus, the Failure Detector can query the NameNode for an updated status report on those related blocks for a given read request. If there is any block failure detected from the report, FastDR will enter the degraded read mode. If the reading process breaks in the middle, the Failure Detector will do necessary rollback and trigger the degraded read mode.

4.2.3 Degraded Read

The Degraded Read Handler takes charge of all degraded read requests. It queries the Cost Measurement Module for the cost vector of the involved DataNodes, executes the degraded read algorithm based on the cost vector to determine the data and parity blocks that need to be downloaded, and instructs the read threads to download the data and parity nodes from the DataNodes in parallel. After receiving all data and parity blocks, it calculates the decoding matrix and reconstructs the lost data.

4.3 More Implementation Details

Our implementation of FastDR is based on HDFS release 0.22.0 and its HDFS-RAID extension [14]. We implement Cauchy Reed Solomon (CRS) codes [4] as a representative of XOR-based erasure codes, by modifying HDFS-RAID's erasure coding layer. We further extend the FSDataInputStream class to support the parallel read operation.

5 Experiments

We first use simulations to extensively evaluate the efficiency of our EG algorithm. To further validate its practicality, we evaluate our FastDR prototype in a practical HDFS cluster testbed. By capturing the actual read/write performance using real storage devices, we aim to show the actual improvement of FastDR over the basic degraded read approach as described in Section 3.3.

Since single-node failures account for the majority of failure patterns in distributed storage systems [17], [19], we assume only a single node failure in our evaluations. Also, we set the parameter $d = n - 1$, meaning that all surviving storage nodes are considered for fulfilling degraded reads. An important future work is to evaluate the impact of the parameter d on the performance of degraded reads.

5.1 Simulations

We now evaluate the efficiency of our EG algorithm. We evaluate the computational overhead and the degraded read performance of three approaches described in Sections 3.3 and 3.4: the basic approach, the enumeration approach, and our EG algorithm. Our goal is to show that our EG algorithm can return near-optimal degraded read solutions in a timely manner.

The degraded read operation performs two steps: (i) reading data blocks and parity blocks from the surviving storage nodes, and (ii) reconstructing the normal blocks and the lost blocks. In our simulation studies, only the running time of the block read part is evaluated. Our justification is that in a distributed environment, the performance bottleneck is due to network transmission instead of computations. Also, previous studies [19], [41] validate that the block read part contributes to the majority of the overall degraded read time.

Our simulations are all conducted under commodity hardware configurations: a Linux-based desktop computer with Intel(R) i3 @3.2GHz CPU and 2GB RAM. The operating system is Ubuntu 12.04. We implement the three degraded read approaches in C.

TABLE 2
Search times of the degraded read solutions for CRS with different configurations of (k, m, w).

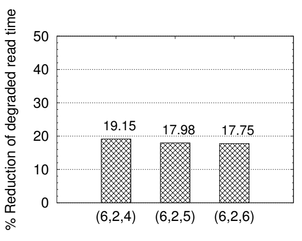
(k,m,w)	Read size	Enumeration's traverse time (s)	EG's traverse time (s)
(12,4,5)	1	1.38	0.096
(12,4,6)	1	37.63	0.151
(12,4,7)	1	754.76	0.217
(10,6,4)	1	19.51	0.298
(10,6,4)	11	19.94	0.412
(10,6,4)	21	22.97	0.671
(10,6,4)	31	23.36	0.907

5.1.1 Search Performance

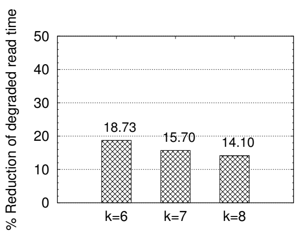
We first compare the enumeration approach and EG on the search times of the degraded read solutions, so as to justify that our EG algorithm can determine its degraded read solution in a very short time, while the enumeration approach is infeasible in practice due to its significantly high computational time. Here, we consider CRS codes [4] and two combinations of (k, m): $(10, 6)$ and $(12, 4)$, both of which are seen in actual deployment [27], [28].

Table 2 shows the search times for CRS codes with different values of w. We observe that the computational time of our EG algorithm is negligible. For all the CRS variants, our EG algorithm can be completed in 1s. On the other hand, the computational time of the enumeration approach increases with nw and also the read size. Also, we find that the computational overhead of the enumeration approach is too expensive. For example, the traverse time is 754s for CRS(12,4,7). Recall that the task of finding the optimal degraded read solution is an on-demand decision. Thus, the enumeration approach is infeasible to fulfill the online decision requirement.

We further evaluate whether EG significantly reduces the degraded read time of the basic approach, and also achieves closely the optimal degraded read time of the enumeration approach. To quantitatively evaluate the degraded read performance for the basic approach and EG, we consider various XOR-based erasure codes that can tolerate different numbers of failures. We consider

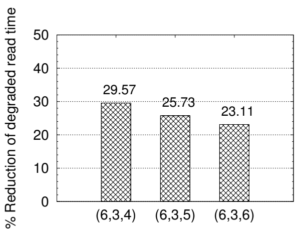


(a) CRS($m = 2$)

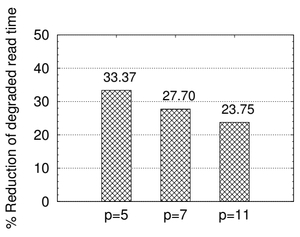


(b) Liber8tion

Fig. 7. Percentage reduction of degraded read time of EG over the basic approach in double-fault tolerant codes.

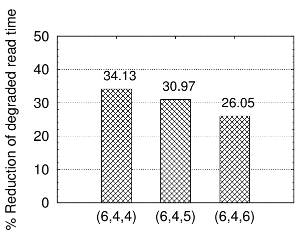


(a) CRS($m = 3$)

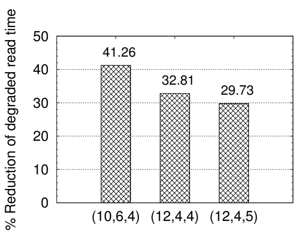


(b) STAR

Fig. 8. Percentage reduction of degraded read time of EG over the basic approach in triple-fault tolerant codes.



(a) CRS($m = 4$)



(b) CRS($m > 4$)

Fig. 9. Percentage reduction of degraded read time of EG over the basic approach in XOR-based erasure codes that can tolerate four or more failures.

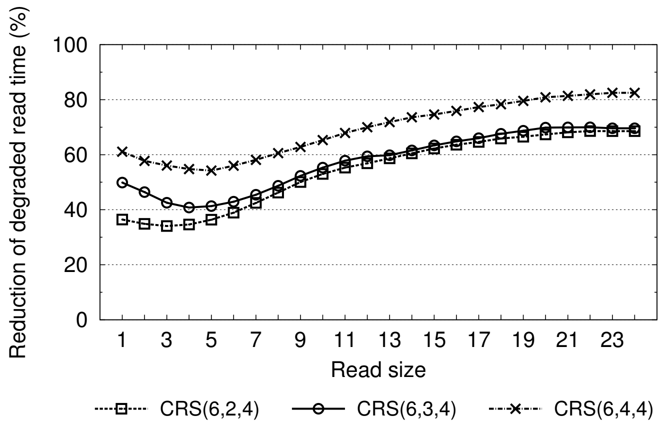


Fig. 10. Percentage reduction of degraded read time of EG over the basic approach in CRS with $m = 2, 3$ and 4.

CRS codes with different settings: $k = 6$ and $m = 2, 3, 4$; $k = 10$ and $m = 6$; and $k = 12$ and $m = 4$. We also consider Liber8tion [25], which belongs to a class of minimum density RAID-6 codes [26] and provides optimal encoding performance. Furthermore, we consider STAR codes [18], which are triple-fault-tolerant.

5.1.2 Degraded Read Performance in Heterogeneous Environments

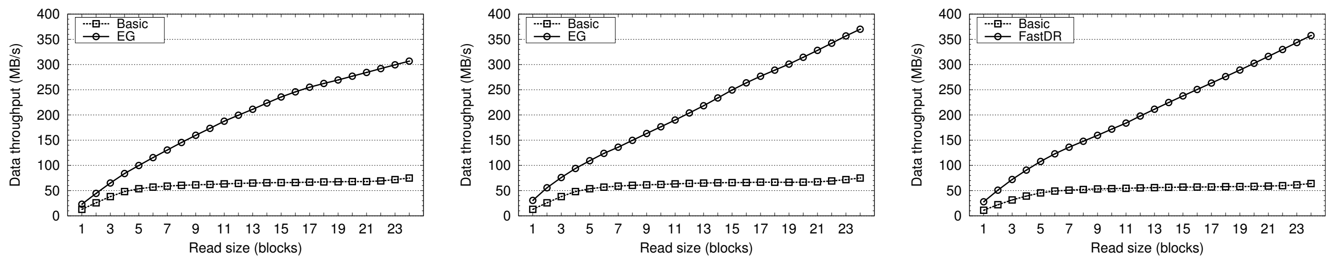
We generate 100 different heterogeneous storage environments for each read size, in which the link transmission bandwidth of each storage node satisfies a uniform distribution U(0.3Mbps, 120Mbps), which has also been used in recent studies to mimic heterogeneous environments for distributed storage [21], [23], [42]. We disable one of the data nodes to represent a single node failure. We then perform the degraded read operation for all possible starting offsets of degraded reads. We only consider the degraded read requests that cover at least one lost block of the failed node, and ignore the read requests that cover only normal blocks. We repeat the evaluations by disabling every data node, and obtain the overall average.

Figures 7, 8, and 9 show the results for different erasure codes. We observe that EG reduces the degraded read time significantly. For example, the percentage reduction of parallel degraded read time is 19.15%, 29.57%, and 34.13% for CRS(6,2,4), CRS(6,3,4), and CRS(6,4,4), respectively. In addition, the improvement of EG over the basic degraded approach becomes more significant when the fault tolerance increases (i.e., m increases). The reason is that XOR-based erasure codes with higher fault tolerance provide more possibilities for searching for an efficient degraded solution. Furthermore, we observe that the gains of EG over the basic approach decrease as w increases. The basic approach downloads the data and parity blocks from k nodes (i.e., $k-1$ surviving data nodes and the first parity node for the case of single failure recovery as assumed in Section 3.3). Therefore, different values of w actually perform the same as the basic approach. However, as w increases, the number of blocks involved in the CDREs of EG increases, and hence the gains of EG decrease.

Due to the very expensive computational overhead, it is infeasible to include the optimal degraded read time of the enumeration approach in Figure 7. Nevertheless, we can still evaluate the degraded read performance for some small sizes of the degraded read request using the enumeration approach and EG. Here we take CRS(6,2,3) as an example. EG reduces the parallel degraded read time by 21.89% ($l = 1$), 23.16% ($l = 2$), 19.69% ($l = 3$), while the reduction of the enumeration approach is 24.05% ($l = 1$), 25.38% ($l = 2$), 23.60% ($l = 3$). Therefore, we believe that EG is robust in returning a near-optimal solution for the degraded read request, while significantly reducing the traverse time.

5.1.3 Degraded Read Performance in the HDFS Environment

We now compare the degraded read performance between the basic approach and EG using the configuration of our HDFS testbed. Our simulation results here



(a) CRS(6,2,4) (b) CRS(6,3,4) (c) CRS(6,4,4)

Fig. 11. Degraded read throughput of the basic approach and EG in CRS with $m = 2, 3$ and 4 (obtained from simulations).

show the maximum possible gains of EG over the basic approach (see Section 5.2). We consider a HDFS cluster with 10 storage nodes (i.e., DataNodes), where the data transmission capability varies from 10MB/s to 80MB/s. We set the block size $\alpha = 1$MB. We have deployed CRS codes with $k = 6$ and $w = 4$ in the cluster. Due to the fact that the number of data blocks in a stripe is 24, our evaluations consider all degraded read requests, with the read size varying from 1 to 24 blocks. Note the fault tolerance m in the system can be configured from 2 to 4. We consider the failure of each node and perform degraded reads for all possible starting offsets. We obtain the average degraded read time for each read size.

Figure 10 plots the degraded read improvement of EG, computed in terms of the average percentage reduction of the degraded read time over the basic approach. For each configuration of CRS codes, Figure 11 also plots the degraded read throughput of the basic approach and EG, defined as the ratio of the read size to the degraded read time. We observe that the gain of EG over the basic approach is significant for each read size. The reason is that EG avoids downloading blocks from the bottlenecked nodes when performing degraded reads in a heterogeneous environment. As the read size increases, the improvement becomes more significant (e.g., by up to 80% when the read size is 20 in CRS(6,4,4) from Figure 10). The reason is that the basic approach needs to read more blocks from the bottlenecked nodes for a larger read size. Furthermore, we see that EG performs better for CRS codes with a higher fault tolerance m.

We see from Figure 10 that when the read size is small (e.g., from 1 to 4), the percentage reduction of EG drops. Recall that the basic approach downloads the data and parity blocks from $k-1$ surviving data nodes and the first parity node for single failure recovery (see Section 3.3). Thus, it reads the $k-1$ contiguous data blocks from $k-1$ surviving data nodes so as to realize degraded reads, which exploits data locality in read requests. This implies that normal blocks in the degraded read request are likely to appear in the collection of contiguous blocks downloaded to reconstruct the lost blocks. When the read size is small, the basic approach does not read additional data blocks, implying that it has roughly the same read performance. On the other hand, FastDR aims to find the set of CDREs that bypass the storage nodes with lower bandwidth. Recall that each read block is associated with a CDRE. When the read size increases at the beginning, the number of CDREs returned also increases, and hence more blocks are to be read and the degraded read time increases. As a result, the percentage reduction of EG over the basic approach drops when the read size is small in Figure 10. Nevertheless, we emphasize that EG still reduces the degraded read time of the basic approach.

5.1.4 Discussion

In this paper, we focus on the degraded reads that request a collection of sequential data blocks. However, in practice, there may be many random access patterns in real storage systems. Therefore, a storage system may respond to a degraded read request that spans different blocks scattered across a stripe. This drop of data locality aggravates the degraded read performance of the basic approach since it reads additional data blocks to reconstruct the lost blocks. Read prefetching can improve the degraded read performance, but its effectiveness depends on the read access patterns. On the other hand, the design of EG is independent of the read access patterns.

5.2 Testbed Evaluations

5.2.1 Methodology

We now conduct testbed experiments on FastDR, which implements the EG algorithm. We also implement the basic approach for our comparison.

Our storage system consists of a cluster of 10 DataNodes, one NameNode, and one FastDR client. All nodes are Linux-based machines with Intel Quad-Core 3.2GHz CPU and 2GB RAM. To mimic node heterogeneity in a practical storage system, each DataNode is equipped with an Ethernet interface card of a network speed that is either 100Mbps or 1Gbps, and a harddisk with different raw read speeds. We interconnect all physical entities over a Gigabit Ethernet switch.

Our experiments mainly evaluate the *data throughput* of degraded reads. We obtain the average data throughput as follows. Before running the experiments, we obtain the related cost vector from the testbed for (see

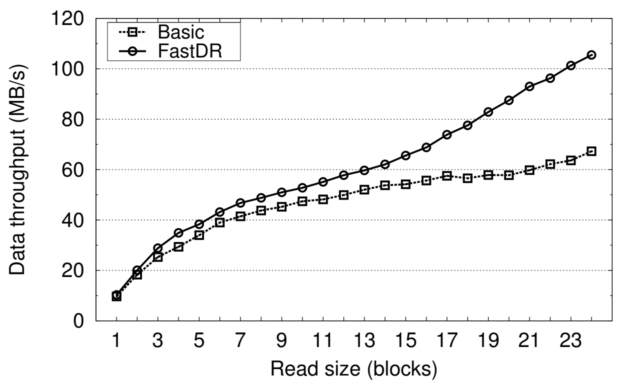


Fig. 12. Experiment 1: Degraded read throughput comparison between the basic approach and FastDR for different read sizes.

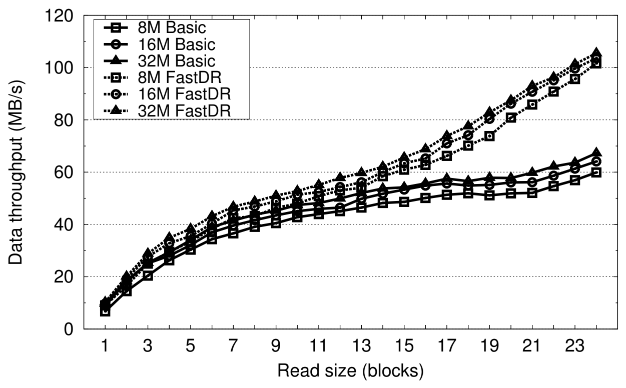


Fig. 13. Experiment 2: Degraded read throughput comparison between the basic approach and FastDR for different block sizes (from 8MB to 32MB).

Section 4.2.1) and include the results as configuration parameters to FastDR. We generate a stripe of data based on the specific coding scheme. We then stripe the data across a collection of DataNodes via HDFS-RAID. We disable one of the DataNodes in the storage system, and then perform degraded read requests to evaluate the performance of degraded reads. Each degraded read operation is evaluated three times. For a specific read size, we repeat this for all possible starting offsets as in previous simulations. We further consider all possible failed nodes, and obtain the overall average. We emphasize that we only focus on degraded requests in our evaluations, and ignore the read requests that cover only normal blocks. For instance, referring to the erasure code (2, 2, 2) in Figure 1, there are two data nodes, and the strip size is two. Thus, we run a total of $2\times2\times3$ degraded read operations for the degraded read request of size 1, and average the data throughput over the 12 runs.

5.2.2 Results

Experiment 1 (Impact of read size on degraded read performance). We first evaluate the impact of read size on the degraded read performance. We consider CRS($k = 6$, $m = 3$, $w = 4$), and vary the degraded read size from 1 to 24. Here, we set the block size to be 32MB.

Figure 12 shows the data throughput of degraded reads versus the read size for the basic approach and FastDR. The data throughput of both approaches increases as the read size increases. The reason is that a degraded read request with a larger read size involves more normal blocks, which are likely to be used directly for reconstructing the lost blocks. Therefore, the data throughput increases as the read size increases, for both the basic approach and FastDR. For example, for the basic approach, the data throughput is 9.67MB/s, 49.92MB/s, 67.27MB/s when the read size is 1, 12, and 24, respectively. Similar results can be observed for FastDR.

We see that the degraded read performance improvement of FastDR over the basic approach becomes more significant as the read size increases. For example, the improvement of FastDR over the basic approach is 6.41%, 15.74% and 56.81% when the read size is 1, 12 and 24, respectively.

Our simulations show that FastDR remarkably improves the time to download the necessary blocks of the basic approach, while our experiments show that FastDR has a smaller performance gain. Recall that the degraded read operation is composed of both read and decoding parts. In order to bypass the bottlenecked nodes in the storage system, FastDR incurs more computational overheads than the basic approach, especially when the degraded read size is small. Thus, FastDR shows a smaller gain. Nevertheless, our experimental results roughly conform to the simulation results shown in Figures 10 and 11.

Experiment 2 (Impact of block size on degraded read performance). We now evaluate the impact of block size on the degraded read performance. Here, we still consider CRS($k = 6$, $m = 3$, $w = 4$), and vary the block size from 8MB to 32MB.

Figure 13 shows the data throughput versus the block size, for the basic approach and FastDR. The data throughput for both approaches increases with the block size. The reason is that with a larger block size, the seek overhead is less and hence both approaches can absorb more of the client's bandwidth. Thus, the degraded read performance can be further improved, although the improvement margin is not significant. Take the basic approach with the read size 1 as an example. Its data throughput with the block size 8MB is 6.76MB/s, while the throughput values for the block size 16MB and 32MB are 8.91MB/s, and 9.67MB/s, respectively. FastDR shows higher data throughput than the basic approach. Note that the improvement is consistent over all read sizes as in the previous experiment.

Experiment 3 (Degraded read performance with different numbers of parity nodes). We now evaluate the degraded read performance when the number of parity nodes is different. We set the block size to be 32MB, and consider CRS(6, 2, 4), CRS(6, 3, 4), and CRS(6, 4, 4).

Figure 14 shows the results. We observe that the basic approach performs roughly the same for CRS with different numbers of parity nodes. The results match the mechanism of the basic approach, which always read the parity block from the first parity node for the degraded read request (see Section 3.3). We also observe from Figure 14 that FastDR performs better for CRS with a larger m. The results closely match those of Figures 10 and 11. For a larger m, the blocks downloaded for degraded reads are distributed over more surviving storage nodes. Thus, the degraded read performance can be improved even more.

Experiment 4 (Performance of degraded reads based on MapReduce). We now evaluate the performance of MapReduce jobs in HDFS-RAID deployed with the basic approach and FastDR in the presence of failures. Here, we select a dataset of size 768MB that contains a collection of English novels on the Project Gutenberg website (http://www.gutenberg.org/). We adopt CRS(6, 3, 4) as the erasure coding scheme, and store the selected dataset in HDFS-RAID. We then run three MapReduce applications: (i) WordCount, which computes the occurrence frequency of each word in the dataset; (ii) Dedup, which removes duplicate lines in the dataset and outputs all unique lines; and (iii) Grep, which extracts matching strings from text files and counts their occurrences.

We emphasize here that a MapReduce job is composed of four parts: Setup, Map, Reduce, and Cleanup, among which only the task of Map involves degraded reads. Therefore, FastDR mainly improves the Map tasks. It also brings benefits for execution of Reduce tasks as the Map tasks can return the intermediate results faster.

Figure 15(a) shows the execution time for WordCount. The percentage reduction of execution time of FastDR over the basic approach is 27.57%, 17.39%, 12.23% with 1, 2, 4 reducers, respectively. Compared to the basic approach, FastDR introduces more read blocks to respond to a degraded read request, which results in more time-consuming coordination between Map tasks and Reduce tasks. Thus, while the addition of the number of reducers improves the execution time of the basic approach, it does not bring benefits for FastDR. Therefore, the improvement of FastDR over the basic approach decreases.

Furthermore, Figures 15(b) and 15(c) show the execution time for Dedup and Grep, respectively. We again observe that FastDR always performs better than the basic approach for different numbers of reducers. For example, our FastDR reduces execution time of the basic approach, by 17.323%, and 18.561% for Dedup and Grep applications, respectively, when using only one reducer.

6 Related Work

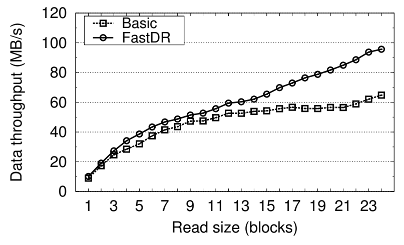
There have been extensive studies on improving the recovery performance of erasure-coded storage systems. Holland *et al.* [16] propose workflow parallelization to speed up reconstruction. FARM [38] improves recovery in large-scale storage systems using parity declustering. Some studies [31], [33], [36] leverage workload characteristics to improve the reconstruction performance.

Several studies minimize recovery I/Os (i.e., the amount of data read from surviving disks) for specific erasure codes. Optimal recovery schemes [34], [37], [39] have been proposed for different RAID-6 codes, and achieve around 25% of I/O savings compared to simply reconstructing all original data. Greenan *et al.* [12] propose new non-MDS XOR-based codes to achieve efficient recovery. Khan *et al.* [19] show that the problem of minimizing recovery I/Os for general erasure codes is NP-hard. Zhu *et al.* [41] propose a greedy recovery heuristic to minimize I/Os for any erasure code. Our approach is built on the greedy heuristic in [41], and extends the latter by taking into account node heterogeneity and I/O parallelism in our optimization model.

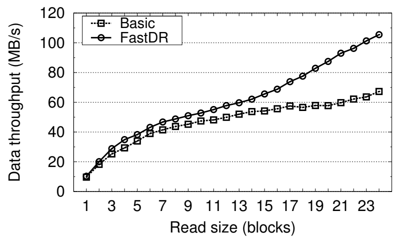
Since nodes are heterogeneous in practice, Greenan *et al.* [13] address data placement of XOR codes on heterogeneous nodes to improve reliability. Zhu *et al.* [42] propose a cost-based recovery heuristic for RAID-6 codes. Our work addresses the heterogeneous systems that employ arbitrary erasure codes.

Besides minimizing I/Os, Dimakis *et al.* [8] propose regenerating codes that minimize the amount of data transfer in distributed storage systems. Note that regenerating codes introduce more I/Os as they read all stored data and transfer the encoded data. Li *et al.* [24] build a system on HDFS that augments existing optimal regenerating codes to support a general number of failures, and demonstrate the saving of data transfer of regenerating codes over erasure codes in recovery.

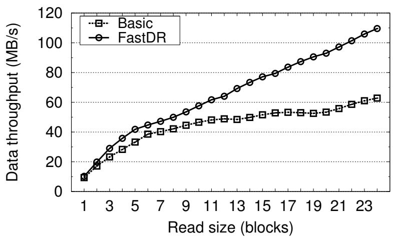
As mentioned in Section 2, degraded reads have different characteristics from recovery of permanent node failures. Existing studies (e.g., [34], [37], [39], [41], [42]) focus on the latter case, and do not account for the read sequence. Khan *et al.* [19] propose the rotated Reed-Solomon (RS) code that reduces I/Os in degraded reads over traditional RS codes. The rotated RS code is designed for sequential read and takes into account the read size (see Section 2). However, it operates by enumerating all decoding equations to find the optimal degraded read solution with minimum I/Os (see Section 3.3). This approach becomes infeasible when we account for the cost of available resources of storage nodes. Huang *et al.* [17] propose a new non-MDS local reconstruction code that minimizes I/Os in degraded reads by adding more parity blocks. Recently, other local reconstruction codes are proposed for achieving efficient recovery [9], [29]. These codes are also evaluated on HDFS. Our work on degraded reads differs from the previous studies in two aspects: (i) instead of proposing a new code construction, we optimize degraded reads for existing erasure codes, and (ii) we address heterogeneous storage systems and identify the effective solution with minimal search time.



(a) CRS($k = 6$, $m = 2$, $w = 4$)

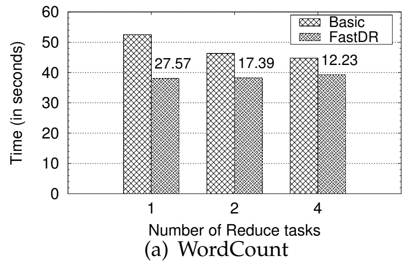


(b) CRS($k = 6$, $m = 3$, $w = 4$)

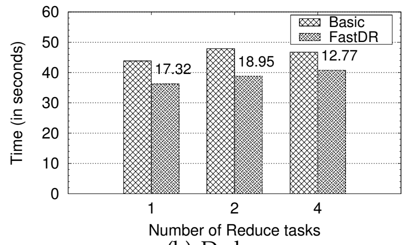


(c) CRS($k = 6$, $m = 4$, $w = 4$)

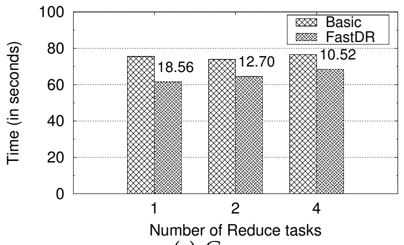
Fig. 14. Experiment 3: Degraded read throughput comparison between the basic approach and FastDR for different levels of fault tolerance (from m =2 to 4).



(a) WordCount



(b) Dedup



(c) Grep

Fig. 15. Experiment 4: MapReduce execution times of the basic approach and FastDR for different types of jobs, including WordCount, Dedup and Grep.

7 Conclusions

Degraded reads have become performance critical operations, due to the fact that temporary errors account for the majority of failures in modern storage systems. To boost the performance of degraded reads in practical erasure-coded storage systems, it is necessary to take into account parallel I/Os and node heterogeneity when performing degraded reads. In addition, as the parameters of degraded reads vary, it is also crucial that the degraded read solution should be found in a timely manner. In this paper, we propose FastDR, a system for boosting degraded reads in XOR-based erasure coded storage systems with heterogeneous storage nodes. We formulate an optimization model, and further propose an enumerated greedy (EG) algorithm to quickly find an efficient degraded read solution in heterogeneous erasure-coded storage systems. Through extensive simulations and testbed experiments, we justify the effectiveness of FastDR in achieving efficient degraded reads in a heterogeneous storage environment. The source code of our FastDR prototype on HDFS is available for download at **http://ansrlab.cse.cuhk.edu.hk/software/fastdr**.

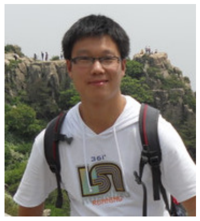
Acknowledgments

The work was supported in part by grants: (i) National Nature Science Foundation of China under Grant No. 61379038, (ii) AoE/E-02/08 and ECS CUHK419212 from the University Grants Committee of Hong Kong, and (iii) Group Research Scheme of CUHK (project no.: 3110110).

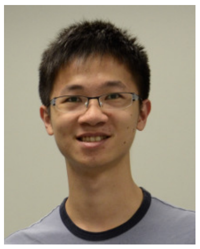
References

[1] M. Abd-El-Malek, W. Courtright II, C. Cranor, G. Ganger, J. Hendricks, A. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. Sambasivan, et al. Ursa Minor: Versatile Cluster-based Storage. In *Proc. of USENIX FAST*, Dec 2005.
[2] 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 *Proceedings of ACM EuroSys*. ACM, 2011.
[3] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of NSDI*, 2004.
[4] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-Based Erasure-Resilient Coding Scheme, 1995.
[5] B. Calder et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, 2011.
[6] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging Endpoint Flexibility in Data-Intensive Clusters. In *Proc. of ACM SIGCOMM*, 2013.
[7] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
[8] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, 2010.
[9] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. The CORE Storage Primitive: Cross-Object Redundancy for Efficient Data Repair & Access in Erasure Coded Storage. In *arXiv preprint arXiv:1302.5192*, 2013.
[10] D. Ford, F. Labelle, F. Popovici, M. Stokely, V. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, 2010.
[11] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, 2003.
[12] K. Greenan, X. Li, and J. Wylie. Flat XOR-based Erasure Codes in Storage Systems: Constructions, Efficient Recovery, and Tradeoffs. In *Proc. of IEEE MSST*, 2010.
[13] K. Greenan, E. Miller, and J. Wylie. Reliability of Flat XOR-based Erasure Codes on Heterogeneous Devices. In *Proc. of IEEE DSN*, 2008.
[14] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.

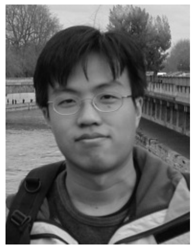
[15] M. Holland, G. Gibson, and D. Siewiorek. Fast, On-line Failure Recovery in Redundant Disk Arrays. In *Proc. of IEEE FTCS*, 1993.
[16] M. Holland, G. Gibson, and D. Siewiorek. Architectures and Algorithms for On-Line Failure Recovery in Redundant Disk Arrays. *Distributed and Parallel Databases*, 2(3):295–335, 1994.
[17] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure Coding in Windows Azure Storage. In *Proc. of USENIX ATC*, 2012.
[18] C. Huang and L. Xu. STAR: An Efficient Coding Scheme for Correcting Triple Storage Node Failures. *IEEE Trans. on Computers*, 57(7):889–901, 2008.
[19] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, 2012.
[20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. Oceanstore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ACM ASPLOS*, 2000.
[21] S. Lee, P. Sharma, S. Banerjee, S. Basu, and R. Fonseca. Measuring Bandwidth Between PlanetLab Nodes. In *Proc. of PAM*, 2005.
[22] J. Li, S. Yang, and X. Wang. Building Parallel Regeneration Trees in Distributed Storage Systems with Asymmetric Links. In *Proc. of CollaborateCom*, pages 1–10. IEEE, 2010.
[23] J. Li, S. Yang, X. Wang, and B. Li. Tree-Structured Data Regeneration in Distributed Storage Systems with Regenerating Codes. In *Proc. of IEEE INFOCOM*, 2010.
[24] R. Li, J. Lin, and P. P. Lee. CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems. In *Proc. of IEEE MSST*, 2013.
[25] J. Plank. A New Minimum Density RAID-6 Code with A Word Size of Eight. In *Network Computing and Applications, 2008. NCA'08. Seventh IEEE International Symposium on*, pages 85–92. IEEE, 2008.
[26] J. Plank, A. Buchsbaum, and B. Vander Zanden. Minimum Density RAID-6 Codes. *ACM Transactions on Storage (TOS)*, 6(4):16, 2011.
[27] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, 2009.
[28] J. K. Resch and J. S. Plank. AONT-RS: Blending Security and Performance in Dispersed Storage Systems. In *Proc. of USENIX FAST*, 2011.
[29] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring Elephants: Novel Erasure Codes for Big Data. In *Proceedings of the VLDB Endowment*, 2013.
[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, pages 1–10. IEEE, 2010.
[31] M. Sivathanu, V. Prabhakaran, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Improving Storage System Availability with D-GRAID. *ACM Transactions on Storage (TOS)*, 1(2):133–170, 2005.
[32] P. Sobe and K. Peter. Flexible Parameterization of XOR based Codes for Distributed Storage. In *Network Computing and Applications, 2008. NCA'08. Seventh IEEE International Symposium on*, pages 101–110. IEEE, 2008.
[33] L. Tian, D. Feng, H. Jiang, K. Zhou, L. Zeng, J. Chen, Z. Wang, and Z. Song. PRO: A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems. In *Proc. of USENIX FAST*, 2007.
[34] Z. Wang, A. Dimakis, and J. Bruck. Rebuilding for Array Codes in Distributed Storage Systems. In *IEEE GLOBECOM Workshops*, 2010.
[35] H. Weatherspoon and J. D. Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *Proc. of IPTPS*, Mar 2002.
[36] S. Wu, H. Jiang, D. Feng, L. Tian, and B. Mao. WorkOut: I/O Workload Outsourcing for Boosting RAID Reconstruction Performance. In *Proc. of USENIX FAST*, 2009.
[37] L. Xiang, Y. Xu, J. Lui, Q. Chang, Y. Pan, and R. Li. A Hybrid Approach to Failed Disk Recovery Using RAID-6 Codes: Algorithms and Performance Evaluation. *ACM Trans. on Storage*, 7(3):11, 2011.
[38] Q. Xin, E. Miller, and S. Schwarz. Evaluation of Distributed Recovery in Large-Scale Storage Systems. In *Proc. of HPDC*, 2004.
[39] S. Xu, R. Li, P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. Lui. Single Disk Failure Recovery for X-code-based Parallel Storage Systems. *IEEE Trans. on Computers*, 63(4):995–1007, 2014.
[40] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proc. of USENIX OSDI*, 2008.
[41] Y. Zhu, P. Lee, Y. Hu, L. Xiang, and Y. Xu. On the Speedup of Single-Disk Failure Recovery in XOR-Coded Storage Systems: Theory and Practice. In *Proc. of IEEE MSST*, 2012.
[42] Y. Zhu, P. Lee, L. Xiang, Y. Xu, and L. Gao. A Cost-based Heterogeneous Recovery Scheme for Distributed Storage Systems with RAID-6 Codes. In *Proc. of IEEE DSN*, 2012.



Yunfeng Zhu received his B.S. from the School of Computer Science, University of Science and Technology of China, Anhui, China, in 2008. He is currently working toward the Ph.D. degree at the School of Computer Science and Technology, University of Science and Technology of China, Hefei, China. His research interests include distributed storage system, cloud storage and data deduplication.



Jian Lin received his B.Eng. in Mathematics and Information engineering and M.Phil. in Computer Science and Engineering from the Chinese University of Hong Kong in 2011 and 2013, respectively. He is now a software developer at Epic. His research interests are in storage systems and distributed systems.



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

Yinlong Xu received his B.S. in Mathematics from Peking University in 1983, and MS and Ph.D in Computer Science from University of Science and Technology of China(USTC) in 1989 and 2004 respectively. He is currently a professor with the School of Computer Science and Technology at USTC. Prior to that, he served the Department of Computer Science and Technology at USTC as an assistant professor, a lecturer, and an associate professor. Currently, he is leading a group of research students in doing some networking and high performance computing research. His research interests include network coding, wireless network, combinatorial optimization, design and analysis of parallel algorithm, parallel programming tools, etc. He received the Excellent Ph.D Advisor Award of Chinese Academy of Sciences in 2006.