Enabling Concurrent Failure Recovery for Regenerating-Coding-Based Storage Systems: From Theory to Practice

Runhui Li, Jian Lin and Patrick P. C. Lee

Abstract—Data availability is critical in distributed storage systems, especially when node failures are prevalent in real life. A key requirement is to minimize the amount of data transferred among nodes when recovering the lost or unavailable data of failed nodes. This paper explores recovery solutions based on regenerating codes, which have been designed to provide fault-tolerant storage and minimum bandwidth. Existing optimal regenerating codes are designed for single node failures. We build a system called CORE, which augments existing optimal regenerating codes for the recovery of a general number of failures including single and concurrent failures. We show theoretically that CORE achieves the minimum possible bandwidth for most cases. We implement a CORE prototype and evaluate it atop an HDFS cluster testbed with up to 20 storage nodes. We demonstrate that our CORE prototype conforms to our theoretical findings and achieves bandwidth savings when compared to the conventional recovery approach based on erasure codes.

Keywords—regenerating codes, failure recovery, distributed storage systems, coding theory, experiments and implementation

1 Introduction

To provide high storage capacity, large-scale distributed storage systems have been widely deployed in enterprises, such as Google File System [15], Amazon Dynamo [10], and Microsoft Azure [5]. In such systems, data is striped across multiple nodes (or servers) that offer local storage space. Nodes are interconnected over a networked environment, in the form of either clustered or wide-area settings.

Ensuring data availability in distributed storage systems is critical, given that node failures are prevalent [15]. Data availability can be achieved via *erasure codes* (e.g., Reed-Solomon codes [41]), whose main idea is to encode data segments into parity segments, such that a subset of the data and parity segments can sufficiently reconstruct the original data segments. Erasure codes can tolerate multiple failures, while incurring less storage overhead than replication.

In addition to tolerating failures, another crucial availability requirement is to *recover* any lost or unavailable data of failed nodes. Recovery is performed in two scenarios: (i) when the failed nodes are crashed and the permanently lost data need to be restored on new nodes, and (ii) when the unavailable data needs to be accessed by clients before the failures are restored. The conventional recovery approach, which applies to *any* erasure code, first reconstructs all original data to obtain the lost/unavailable data. Since the lost/unavailable data usually accounts for only a fraction of the original data, previous studies explore how to optimize the recovery performance by minimizing the amount of data communicated. One class of approaches is to minimize I/Os (i.e., the amount of data read from disks) based on erasure codes (e.g., [26], [30], [42], [53], [55]). Another class of approaches is to minimize the bandwidth (i.e., the amount of data transferred over a network during recovery) based on *regenerating codes* [11], in which each surviving node encodes its stored data and sends encoded data for recovery. In the scenario where network capacity is limited, minimizing the bandwidth can improve the overall recovery performance. In this work, we explore the feasibility of deploying regenerating codes in practical distributed storage systems.

Most existing recovery approaches, including those for minimizing I/Os and bandwidth, are restricted to *single failure* recovery. Although single failures are common in distributed storage systems [39], node failures are often correlated and co-occurring in practice, as reported in both clustered storage (e.g., [14], [43]) and wide-area storage (e.g., [6], [20], [33]). To provide tolerance against *concurrent* (multiple) failures, data is usually protected with a high degree of redundancy. For example, Cleversafe [7], a commercial wide-area storage system, encodes every 10 data segments into 6 parity segments [36]. Some wide-area storage systems, such as OceanStore [31] and CFS [8], employ erasure codes with double redundancy (i.e., twice the original storage space).

Thus, we believe that minimizing the bandwidth for concurrent failure recovery provides additional benefits for today's large-scale distributed storage systems. For example, concurrent failure recovery is beneficial to delaying immediate recovery [2], [50]. That is, we can perform recovery only when the number of failures ex-

- *R. Li, J. Lin and P. Lee are with the Chinese University of Hong Kong, Shatin, N.T., Hong Kong (emails: {rhli, jlin, pclee}@cse.cuhk.edu.hk)*
- *A 6-page shorter conference version of this paper appeared in Proceedings of the 29th IEEE Conference on Massive Data Storage (MSST), May 2013 [32]. In this journal version, we include additional results of our analysis and testbed experiments on CORE.*

ceeds a tolerable limit. This avoids unnecessary recovery should a failure be transient and the data be available shortly (e.g., after rebooting a failed node). Given the importance of concurrent failure recovery, we thus pose the following questions: (1) Can we achieve bandwidth saving, based on regenerating codes, in recovering a general number of failures including single and concurrent failures? (2) If we can enable regenerating codes to recover concurrent failures, can we seamlessly integrate the solution into a practical distributed storage system?

In this paper, we propose a complete system called CORE, which supports both single and concurrent failure recovery and aims to minimize the bandwidth of recovering a *general* number of failures. CORE augments existing optimal regenerating codes (e.g., [37], [51]), which are designed for single failure recovery, to also support concurrent failure recovery. A key feature of CORE is that it retains existing optimal regenerating code constructions and the underlying regenerating-coded data. That is, instead of proposing new code constructions, CORE adds a new recovery scheme atop existing regenerating codes. Its main idea is to download the same data as that would have been downloaded under the existing regenerating codes for recovery of each of the failed nodes and show that all the failed nodes can be concurrently recovered.

This work studies both the theoretical and applied aspects of CORE. On the theoretical side, we show that CORE achieves the minimum bandwidth for a majority of concurrent failure patterns. We also propose extensions to CORE to achieve sub-optimal bandwidth saving even for the remaining concurrent failure patterns. Our analytical study validates that CORE can recover concurrent failure patterns with significant bandwidth savings over conventional recovery based on erasure codes. For example, for (20,10), the bandwidth savings are 36-64% and 25-49% in the optimal and sub-optimal cases, respectively. We also show via analysis that CORE has significantly higher durability than conventional recovery.

On the applied side, we implement a prototype of CORE and demonstrate the feasibility of deploying CORE, or regenerating codes in general, in a practical distributed storage system. As a proof of concept, we choose the Hadoop Distributed File System (HDFS) [49] as a starting point. CORE sits as a layer atop HDFS and supports recovery for a general number of failures. We also adopt a pipelined implementation that parallelizes and speeds up the recovery process. We extensively experiment CORE on an HDFS testbed with up to 20 storage nodes. We show that compared to erasure codes, CORE achieves recovery throughput gains with up to $3.4\times$ for single failures and up to $2.3\times$ for concurrent failures. Our experimental results conform to our theoretical findings. We also evaluate the runtime performance of MapReduce jobs under node failures. We note that there remain very limited studies on the practical deployment of regenerating codes in the literature. Our CORE implementation provides insights into the feasibility of regenerating-coding-based recovery in improving the availability of a practical storage system.

The rest of the paper proceeds as follows. Section 2 formulates our system model. Section 3 presents the design of CORE and our analytical findings. Section 4 describes the implementation details of CORE. Section 5 presents experimental results. Section 6 reviews related work. Section 7 discusses open issues, and finally, Section 8 concludes this paper. We also refer readers to our digital supplementary file for additional analysis.

2 System Model

2.1 Basics

We first define the terminologies and notations. Table 1 summarizes the major notations used in this paper. We consider a distributed storage system composed of a collection of *nodes*, each of which refers to a physical storage device. The storage system contains n nodes labeled by $N_0, N_1, \cdots, N_{n-1}$, in which k nodes (called *data nodes*) store the original (uncoded) data and the remaining $n-k$ nodes (called *parity nodes*) store parity (coded) data. The coding structure is *systematic*, meaning that the original data is kept in storage.

Figure 1 shows an example of a distributed storage system, which is also consistent with the design of the erasure-coded module of HDFS called HDFS-RAID [22]. Each node stores a number of *segments*. A segment is the basic unit of read/write operations in a storage system. It is called a *data segment* if it holds original data, or a *parity segment* if it holds parity data. To store data/parity information, we partition the stored data into collections of n segments that comprise k data segments and $n-k$ parity segments. To generate parity segments, each data (parity) segment is partitioned into fixed-size data (parity) *strips* of r *packets* each, such that a packet is the basic unit of encoding/decoding operations. Each parity strip is encoded from the k data strips, and a *stripe* is composed of k data strips and $(n-k)$ parity strips. Encoding is done on a per-stripe basis. A data (parity) segment contains all data (parity) strips, so each collection of n segments comprises multiple stripes. The n segments in each collection are distributed across n nodes. For load balancing reasons the identities of the data/parity nodes are rotated so that the data and parity segments are evenly distributed across nodes [30], [36].

Each stripe is independently encoded. Our discussion thus focuses on a single stripe and our recovery scheme will operate on a per-stripe basis. Let M be the total amount of original uncoded data stored in a stripe. Let $s_{i,j}$ be a stored packet of node N_i at offset j in a stripe, where $i = 0, 1, \cdots, n-1$ and $j = 0, 1, \cdots r-1$. Each stripe contains nr stored packets, which can be formed by multiplying an $nr \times kr$ *generator matrix* by a vector of kr original data packets based on *Galois field* arithmetic, whose implementation details can be found in the prior study [17]. This work focuses on the arithmetic over

TABLE 1
Major notations used in this paper.

n	number of nodes
N_i	the i-th node ($0 \leq i \leq n-1$)
k	number of data nodes
r	number of packets per strip
t	number of concurrent failures ($1 \leq t \leq n-k$)
M	size of original data stored in a stripe
$s_{i,j}$	the j-th stored packet in a stripe of node N_i ($0 \leq i \leq n-1$, $0 \leq j \leq r$)
$e_{i,i'}$	encoded packet from surviving node N_i to decode lost data of failed node $N_{i'}$ ($0 \leq i, i' \leq n-1$)

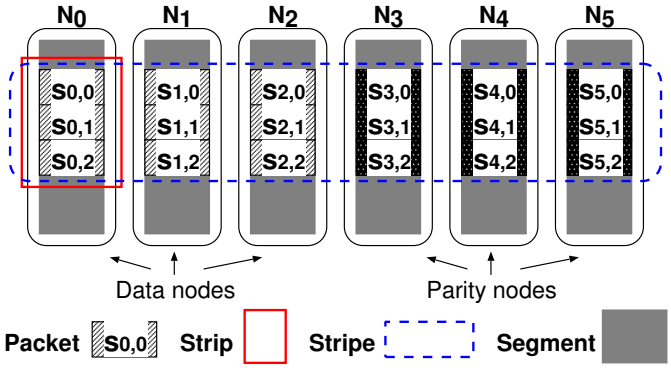


Fig. 1. Example of a distributed storage system, where $n = 6$, $k = 3$, and $r = 3$. Nodes N_0, N_1, and N_2 are data nodes, while N_3, N_4, and N_5 are parity nodes.

the Galois field GF(2^8). Note that our recovery scheme applies to the failures of both data and parity nodes. It treats each stored packet $s_{i,j}$ the same way regardless of whether it is a data or parity packet.

For data availability, we have the storage system employ an (n, k) erasure code that is *maximum distance separable (MDS)*, meaning that the stored data of any k nodes can be used to reconstruct the original data. That is, an (n, k) MDS-coded storage system can tolerate any $n-k$ out of n concurrent failures. MDS codes also ensure optimal storage efficiency, such that each node stores $\frac{M}{k}$ units of data per strip. Reed-Solomon (RS) codes [41] are a classical example of MDS codes. RS codes can be implemented with strip size $r = 1$ to minimize the generator matrix size.

2.2 Recovery

Our recovery addresses two types of node failures. The first type is the recovery from permanent failures (e.g., due to crashes) where data is permanently lost. In this case, we reconstruct the lost data of the failed nodes on new nodes to minimize the window of vulnerability. Another type is degraded reads to the temporarily unavailable data during transient failures (e.g., due to system reboots or upgrades) or before the permanent failures are restored. The reads are degraded as the unavailable data needs to be reconstructed from the available data of other surviving nodes. In our discussion, we use "lost data" to refer to both permanently lost data and temporarily unavailable data.

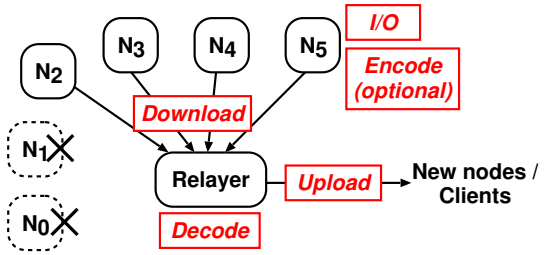


Fig. 2. Recovering nodes N_0 and N_1 using the relayer.

The storage system activates recovery of lost data when there are a number $t \geq 1$ of failed nodes. Clearly, we require $t \leq n-k$, or the original data will be unrecoverable. We call the set of t failed nodes the *failure pattern*. The lost data will be reconstructed by the data stored in other surviving nodes.

Our recovery builds on the *relayer* model, in which a relayer daemon coordinates the recovery operation. Figure 2 depicts the relayer model. During recovery, each surviving node performs two steps: (i) *I/O*: it reads its stored data, and (ii) *encode*: it combines the stored data into some linear combinations and is necessary for some erasure code constructions (see Section 2.3). The relayer daemon performs three steps: (i) *download*: it downloads the data from some other surviving nodes, (ii) *decode*: it decodes the lost data using the downloaded data, and (iii) *upload*: it uploads the decoded data to the new nodes (for recovery from permanent failures) or to the client who requests the data (for degraded reads). We assume that the relayer is reliable during the recovery process. Note that this relayer model is used in prior studies in the contexts of peer-to-peer storage [2], data center storage [26], and proxy-based cloud storage [23].

To improve the recovery performance of a distributed storage system with limited network bandwidth, it is important to minimize the amount of data transferred over the network. If the number of failed nodes is small, the amount of data being downloaded from the surviving nodes is larger than the amount of decoded data being uploaded to new nodes or clients. If we pipeline the download and upload steps (see Section 4.2), then the download step becomes the bottleneck. Thus, we focus on optimizing the download step in recovery. Formally, we define the *recovery bandwidth* as the total amount of data being downloaded per stripe from the surviving nodes to the relayer during recovery. Our goal is to minimize the recovery bandwidth.

2.3 Regenerating Codes

When a conventional erasure-coded system detects failures, the relayer downloads data from any k surviving nodes and recovers the lost data. We refer to this recovery process as *conventional recovery*. The amount of data being downloaded in conventional recovery is

equal to the amount of original data being stored (i.e., M per stripe). Note that some erasure code constructions allow less data to be read (see Section 6). However, conventional recovery applies to *any MDS erasure code and any number of failures no more than* $n-k$. In this paper, when we refer to erasure codes, we assume that conventional recovery is used.

We consider a special class of codes called *regenerating codes* [11] that enables the relayer to download less than the amount of original data being stored. Regenerating codes build on network coding [1], in which during recovery, surviving nodes compute and send encoded packets to the relayer, which then decodes the lost data using the encoded packets. Each encoded packet is formed by a linear combination of the stored packets in a surviving node (note that it may be identical to one of the stored packets). It is shown that regenerating codes lie on an optimal tradeoff curve between storage cost and recovery bandwidth [11]. There are two extreme points: *minimum storage regenerating (MSR)* codes, in which each node stores the minimum amount of data on the tradeoff curve, and *minimum bandwidth regenerating (MBR)* codes, in which the bandwidth is minimized. Note that MSR codes have the optimal storage efficiency and are MDS (see Section 2.1), while MBR codes minimizes bandwidth at the expense of higher storage overhead. In this work, we focus on MSR codes.

Existing optimal MSR codes are designed for recovering single failures, as described below. First, the strip has $r=n-k$ packets to achieve the minimum possible bandwidth. During recovery, the relayer downloads one *encoded* packet from each of the $n-1$ surviving nodes[1]. Let $e_{i,i'}$ be the encoded packet downloaded from node N_i and used to decode the lost packets of the failed node $N_{i'}$. Each encoded packet $e_{i,i'}$ is a function of the packets $s_{i,0}, s_{i,1}, \cdots, s_{i,r-1}$ stored in the surviving node N_i, and has the same size as each stored packet. Using the encoded packets, the relayer decodes the lost packets of the failed node $N_{i'}$. MSR codes achieve the minimum recovery bandwidth (denoted by γ_{MSR}) for single failure recovery given by [11]:

$$\gamma_{MSR} = \frac{M(n-1)}{k(n-k)}. \quad (1)$$

However, if more than one node fails, the optimal MSR code constructions cannot connect to $n-1$ surviving nodes and achieve the saving shown in Equation (1). To recover concurrent failures, a straightforward approach is to resort to conventional recovery and download the size of original data from any k surviving nodes. This paper explores if we can achieve recovery bandwidth savings for concurrent failures as well.

1. There are MSR code constructions (e.g., [37], [51]) that can download encoded packets from fewer than $n-1$ surviving nodes at the expense of higher recovery bandwidth. In this paper, we mainly focus on the case where $n-1$ surviving nodes are connected. Our digital supplementary file provides additional analysis when the baseline MSR codes connect to fewer than $n-1$ surviving nodes.

3 Design of CORE

CORE builds on existing MSR code constructions that are designed for single failure recovery with parameters (n,k). CORE has two major design goals. First, CORE preserves existing code constructions and stored data. That is, we still have data striped and stored with existing MSR code constructions, while CORE sits as a layer atop existing MSR code constructions and enables efficient recovery for both single and concurrent failures. The optimal storage efficiency of MSR codes is still preserved. Second, CORE aims to minimize recovery bandwidth for a variable number $t \leq n-k$ of concurrent failures, without requiring t to be fixed before a code is constructed and the data is stored.

In this section, we first describe the baseline approach of CORE, in which we extend the existing optimal solution of single failure recovery to support concurrent failure recovery (Section 3.1). We note that the baseline approach of CORE is not applicable in a small fraction of failure patterns, so we propose a simple extension that still provides bandwidth reduction for such cases (Section 3.2). We present theoretical results showing that CORE can reach the optimal point for a majority of failure patterns (Section 3.3). Finally, we analyze the recovery bandwidth savings (Section 3.4) and reliability (Section 3.5) of CORE.

In our digital supplementary file, we present the proofs of the theorems stated in Section 3.3. We also provide additional analysis bandwidth savings of CORE when compared to the baseline MSR codes and immediate recovery.

3.1 Baseline Approach of CORE

We first provide the background of existing MSR code constructions on which CORE is developed. We then define the building blocks of CORE, and explain how CORE uses these building blocks to support concurrent failure recovery.

Background. CORE can build on existing optimal MSR code constructions including Interference Alignment (IA) codes [44], [46], [51] and Product-Matrix (PM) codes [37]. Here, we provide a high-level overview of how IA codes work; PM codes work similarly. IA codes are originally proposed in [46] (subsequently called MISER codes [44]) with the encoding and decoding algorithms for data segments, and the decoding algorithm for parity segments are proposed in [51]. In a high level, IA codes extend the idea of aligning interference signals in wireless communication into failure recovery in distributed storage systems. Recall that each stripe in regenerating codes contains $k(n-k)$ original data packets (see Section 2.1). Each stored packet can be viewed as a linear combination of the $k(n-k)$ original data packets. Suppose that a data node fails (the similar idea also applies for parity nodes). The $n-1$ surviving nodes compute the $n-1$ encoded packets (denoted by $\mathbf{y}=(y_1,\cdots,y_{n-1})^T$). The relayer downloads the $n-1$

encoded packets and decodes the $n-k$ lost data packets (denoted by $\mathbf{x}_1 = (x_1, \cdots, x_{n-k})^T$) of the failed node. There are other $(k-1)(n-k)$ data packets (denoted by $\mathbf{x}_2 = (x_{(n-k)+1}, \cdots, x_{k(n-k)})^T$) that need not be regenerated and thus can be viewed as interference signals. We can express $\mathbf{y}$ as a system of equations in $\mathbf{x}_1$ and $\mathbf{x}_2$ as:

$$(\mathbf{A} \mid \mathbf{B}) \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \mathbf{y},$$

for some coefficient matrices $\mathbf{A}$ and $\mathbf{B}$ of sizes $(n-1) \times (n-k)$ and $(n-1) \times (k-1)(n-k)$, respectively. By elementary row operations, we can transform the system of equations into:

$$\left(\begin{array}{c|c} \mathbf{A}' & \mathbf{0} \\ \mathbf{0} & \mathbf{B}' \end{array} \right) \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \end{pmatrix} = \mathbf{y}',$$

for transformed vector $\mathbf{y}'$ and transformed matrices $\mathbf{A}'$ and $\mathbf{B}'$ of sizes $(n-k)\times(n-k)$ and $(k-1)\times(k-1)(n-k)$, respectively. Note that IA codes ensure that there exists some transformation that makes $\mathbf{A}'$ an invertible matrix, so that $\mathbf{x}_1$ (i.e., the lost packets) can be uniquely solved.

IA codes construct the generator matrix that satisfies the above properties. PM codes have a similar idea using a different generator matrix design. We refer readers to [37], [44], [46], [51] for their mathematical details on the generator matrix design. We point out that constructing erasure codes that achieve interference alignment has been a challenging topic for several years.

Note that both IA and PM codes have parameter constraints. IA codes require $n \geq 2k$, and PM codes require $n \geq 2k-1$. In this work, we mainly focus on the double redundancy $n = 2k$, which is also considered in state-of-the-art distributed storage systems such as OceanStore [31] and CFS [8]. While the redundancy overhead is higher than traditional RAID-5 and RAID-6 codes for large (n,k), it remains less than traditional 3-way replication used in production storage systems such as GFS [15] and HDFS [49].

Building blocks. Our observation is that any optimal MSR code construction can be defined by two functions. Let $\mathsf{Enc}_{i,i'}$ be the encoding function that is called by node N_i to generate an encoded packet $e_{i,i'}$ for the failed node $N_{i'}$ using the $r = n-k$ stored packets in node N_i as inputs; let $\mathsf{Dec}_{i'}$ be the decoding function that returns the set of $n-k$ stored packets of a failed node N_i' using the encoded packets from the other $n-1$ surviving nodes as inputs. Both Enc and Dec define the operations of linear combinations of the stored packets $s_{i,j}$'s, depending on the specific code construction. From the above discussion, Enc is to construct the encoded packets $\mathbf{y}$, while Dec is to decode the lost packets $\mathbf{x_1}$.

CORE works for *any* construction of optimal MSR codes, as long as the functions Enc and Dec are well-defined. The two functions Enc and Dec form the building blocks of CORE.

Main idea of the baseline approach. We consider two types of encoded packets to be downloaded for recovery: *real packets* and *virtual packets*. To recover each of the t failed nodes (where $t > 1$), the relayer still operates as if it connects to $n-1$ nodes, but this time it represents the packets to be downloaded from the failed nodes as virtual packets, while still downloading the packets from the remaining $n-t$ surviving nodes as real packets. Now, using Enc and Dec, we compute each virtual packet as a function of the downloaded real packets. Finally, using the downloaded real packets and the reconstructed virtual packets, we can decode the lost stored packets in the failed nodes.

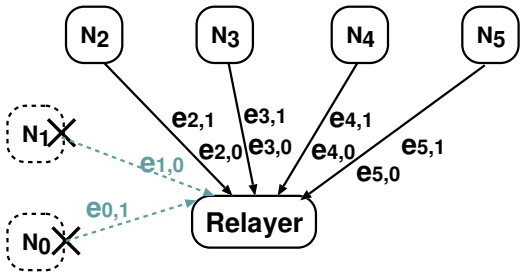


Fig. 3. Example of how the relayer downloads real and virtual packets for a (6,3) code for the recovery of failed nodes N_0 and N_1. Here, $e_{1,0}$ and $e_{0,1}$ are the virtual packets.

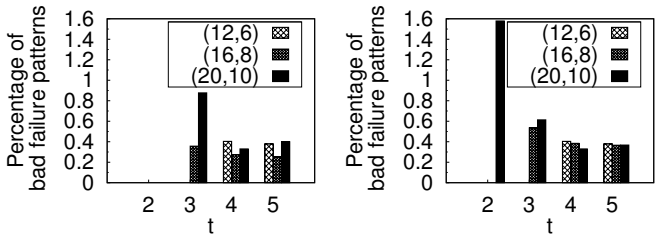
Example. We depict our idea using Figure 3, which shows a (6,3) code and has failures N_0 and N_1. The two encoded packets $e_{1,0}$ and $e_{0,1}$ are virtual packets, and the rest are real packets. We can express $e_{1,0}$ and $e_{0,1}$ based on Enc and Dec for single failure recovery as:

$$\begin{aligned} e_{1,0} &= \mathsf{Enc}_{1,0}(s_{1,0}, s_{1,1}, s_{1,2}) \\ &= \mathsf{Enc}_{1,0}(\mathsf{Dec}_1(e_{0,1}, e_{2,1}, e_{3,1}, e_{4,1}, e_{5,1})) \\ e_{0,1} &= \mathsf{Enc}_{0,1}(s_{0,0}, s_{0,1}, s_{0,2}) \\ &= \mathsf{Enc}_{0,1}(\mathsf{Dec}_0(e_{1,0}, e_{2,0}, e_{3,0}, e_{4,0}, e_{5,0})) \end{aligned}$$

The encoded packet $e_{1,0}$ is computed by encoding the stored packets $s_{1,0}$, $s_{1,1}$, and $s_{1,2}$, all of which can be reconstructed from other encoded packets $e_{0,1}$, $e_{2,1}$, $e_{3,1}$, $e_{4,1}$, and $e_{5,1}$ based on single failure recovery. Thus, $e_{1,0}$ can be expressed as a function of encoded packets. The encoded packet $e_{0,1}$ is expressed in a similar way. Now, we have two equations with two unknowns $e_{1,0}$ and $e_{0,1}$. If these two equations are linearly independent, we can solve for $e_{1,0}$ and $e_{0,1}$. Then we can apply Dec_0 and Dec_1 to decode the lost stored packets of N_0 and N_1. In general, to recover t failed nodes, we have a total of $t(t-1)$ virtual packets. We can compose $t(t-1)$ equations based on the above idea. If these $t(t-1)$ equations are linearly independent, we can solve for the virtual packets. A subtle issue is that the system of equations may not have a unique solution. We explain how we generalize our baseline approach for such an issue in the next subsection.

3.2 Recovering Any Failure Pattern

We aim to express virtual packets as functions of real packets by solving a system of equations. However, we



(a) IA codes (b) PM codes

Fig. 4. Percentages of bad failure patterns for different (n, k) and t.

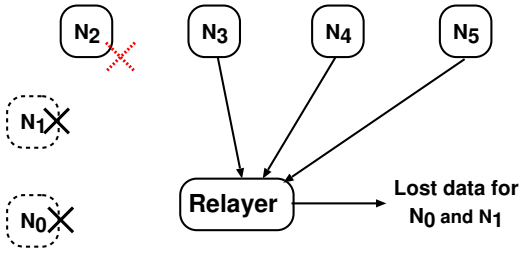


Fig. 5. An example of using a virtual failure pattern for a (6,3) code. If the original failure pattern $\{N_0, N_1\}$ is bad, then we can instead recover the virtual failure pattern $\{N_0, N_1, N_2\}$ and only download encoded packets from nodes N_3, N_4, N_5.

note that for some failure patterns (i.e., the set of failed nodes), the system of equations cannot return a unique solution. A failure pattern is said to be *good* if we can uniquely express the virtual packets as a function of the real packets, or *bad* otherwise. Our goal is to reduce the recovery bandwidth even for bad failure patterns.

We first study the likelihood of having bad failure patterns. However, developing theoretical proofs for quantifying such likelihood is challenging, because the likelihood depends on the code construction. We observe that both IA and PM codes have different likelihoods for the same parameters (n, k) and t. Fortunately, in practical deployment, even though a storage system contains a large number of nodes, the stripe size n is always limited to a reasonably small value to avoid extra coding overhead [30], [36]. Thus, we resort to enumeration, which suffices for practical use cases and enables us to feasibly identify all bad failure patterns in advance. Specifically, given an (n, k) code and t failures, there are $\binom{n}{t}$ possible failure patterns. We enumerate all the possible failure patterns and check if each of them is bad. We conduct our enumeration for both IA and PM codes.

Figure 4 shows the percentages of bad failure patterns for different combinations of (n, k) and t. We observe that among all parameters we consider, bad failure patterns only account for a small percentage, with at most 0.9% and 1.6% for IA and PM codes, respectively. Also, for some sets of parameters, we do not find any bad failure patterns. Nevertheless, we would like to reduce the recovery bandwidth for such bad failure patterns even though they are rare.

We now extend our baseline approach of CORE to deal with the bad failure patterns, with an objective of reducing the recovery bandwidth over the conventional recovery approach. For a bad failure pattern $\mathcal{F}$, we include one additional surviving node and form a *virtual failure pattern* $\mathcal{F}'$, such that $\mathcal{F} \subset \mathcal{F}'$ and $|\mathcal{F}'| = |\mathcal{F}| + 1 = t + 1$. Then the relayer downloads the data from the $n - t - 1$ nodes outside $\mathcal{F}'$ needed for decoding the lost data of $\mathcal{F}'$, although actually only the lost data of $\mathcal{F}$ needs to be decoded. Figure 5 shows an example of how we use a virtual failure pattern for recovery. If $\mathcal{F}'$ is still a bad failure pattern, then we include an additional surviving node into $\mathcal{F}'$, and repeat until a good failure pattern is found. Note that the size of $\mathcal{F}'$ must be upper-bounded by $n - k$, as we can always connect to k surviving nodes to reconstruct the original data.

The number of bad failure patterns depends on the code construction and the parameters (n, k). Reasoning the presence of bad failure patterns remains an open issue and is posed as future work.

3.3 Theoretical Results

We present two theorems. The first one shows the lower bound of recovery bandwidth. The second one shows that CORE achieves the lower bound for good failure patterns.

Theorem 1: Suppose that we recover t failed nodes. The lower bound of recovery bandwidth is:

$$\begin{cases} \dfrac{Mt(n-t)}{k(n-k)} & \text{where } t < k, \\ M & \text{where } t \geq k. \end{cases}$$

□

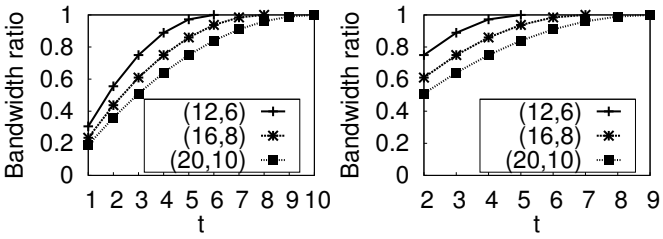
Theorem 2: CORE, which builds on MSR codes for single failure recovery, achieves the lower bound in Theorem 1 if we recover a good failure pattern. □

Since most failure patterns are good (with at least 99.1% and 98.4% for IA and PM codes, respectively), we conclude that CORE minimizes recovery bandwidth for a majority of failure patterns. In the next subsection, we show the actual bandwidth savings of CORE in both good and failure patterns.

3.4 Analysis of Bandwidth Savings

We now study the bandwidth savings of CORE over conventional recovery. We compute the bandwidth ratio, defined as the ratio of recovery bandwidth of CORE to that of conventional recovery. We vary (n, k) and the number t of failed nodes to be recovered.

We first consider good failure patterns. For CORE, the recovery bandwidth achieves the lower bound derived in Theorem 1, and we can directly apply the theoretical results. For conventional recovery, the recovery bandwidth is the amount of original data being stored. Figure 6(a) shows the bandwidth ratios. We observe that CORE achieves bandwidth savings in both single



(a) Good failure patterns (b) Bad failure patterns

Fig. 6. Ratio of recovery bandwidth of CORE to that of conventional recovery.

and concurrent failures. For single failures (i.e., $t = 1$), CORE directly benefits from existing regenerating codes, and saves the recovery bandwidth by 70-80%. For concurrent failures (i.e., $t > 1$), CORE also shows bandwidth savings, for example by 44-64%, 25-49%, and 11-36% for $t = 2$, $t = 3$ and $t = 4$, respectively. The bandwidth savings decrease as t increases, since more lost data needs to be reconstructed and we need to retrieve nearly the amount of original data stored. On the other hand, the bandwidth savings increase with the values of (n, k). For example, the saving is 36-64% in (20,10) when $2 \leq t \leq 4$.

We now study how CORE performs for bad failure patterns. Recall from Section 3.2 for each bad failure pattern $\mathcal{F}$, CORE forms a virtual failure pattern $\mathcal{F}'$ that is a good failure pattern. We compute the recovery bandwidth for $\mathcal{F}'$ based on our theoretical results in Section 3.3. Figure 6(b) shows the bandwidth ratios. We find that in all cases we consider, it suffices to add one surviving node into $\mathcal{F}'$ (i.e., $|\mathcal{F}'| = |\mathcal{F}| + 1$) and obtain a good failure pattern. Thus, the recovery bandwidth of CORE for a bad t-failure pattern is always equivalent to that for a good $(t+1)$-failure pattern. From the figure, we still see bandwidth savings of CORE over conventional recovery. For example, the saving is 25-49% in (20,10) when $2 \leq t \leq 4$.

3.5 Analysis of Reliability

We conduct a reliability analysis on CORE and conventional recovery using a Markov model. Let X be the random variable representing the time elapsed until the data of a storage system becomes unrecoverable. We define the mean-time-to-data-loss (MTTDL) as the expected value of X. We note that although MTTDL has its deficiencies [18], it has been used to analyze the reliability of systems with replication (e.g., [14]) and erasure codes (e.g., [16], [26], [42]). We only use MTTDL for reliability comparisons due to different recovery performance, instead of quantifying the real reliability of a storage system.

Figure 7 shows the Markov model of (n, k) codes. Let state t, where $0 \leq t \leq n - k$, denote that the storage system has t failures, and state $n - k + 1$ denote that the storage system has more than $n - k$ failures and its

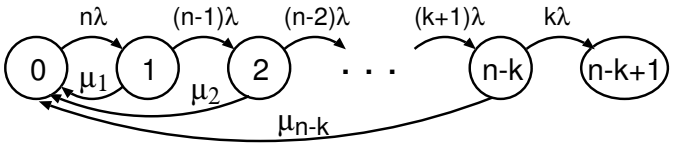


Fig. 7. Reliability model of (n, k) codes.

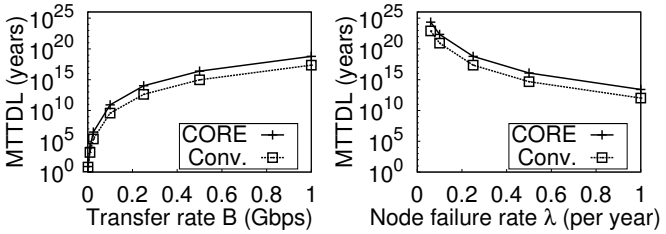
data becomes unrecoverable. To simplify the problem, we assume that node failures occur independently and have constant rates as in prior studies (e.g., [14], [16], [26], [42]). Let λ denote the failure rate of a single node. Thus, the transition rate from state t (where $0 \leq t \leq n-k$) to state $t+1$ is $(n-t)\lambda$. In concurrent recovery (assuming the relayer model in Section 2.2 is used), every state t (where $1 \leq t \leq n - k$) transitions to state 0 at rate μ_t, which depends on the recovery scheme being used. To compute μ_t, let B be the transfer rate of downloading data from surviving nodes for recovery, and S be the storage capacity of a single storage node (i.e., the amount of original data is kS). To recover t failures, CORE downloads $\frac{t(n-t)}{k(n-k)} \times kS$ units of data in most cases (see Theorems 1 and 2)[2] and hence $\mu_t = \frac{(n-k)B}{t(n-t)S}$; conventional recovery downloads kS units of data and hence $\mu_t = \frac{B}{kS}$. Once the Markov model is constructed, we can obtain the MTTDL by calculating the expected time to reach the absorbing state $n-k+1$. We refer readers to [16] for the detailed derivations of the MTTDL.

We use $(n, k) = (16, 8)$ as an example to compare the MTTDLs of CORE and conventional recovery. MTTDL is determined by three variables: storage capacity of each node S, transfer rate B and node failure rate λ. First, we fix the mean failure time $1/\lambda = 4$ years [42] and S =1TB, and evaluate the impact of B on the MTTDLs. Figure 8(a) shows the MTTDL results. With the increasing transfer rate, the recovery rate and hence the MTTDLs of both CORE and conventional recovery increase. Next, we fix B = 1Gbps and S =1TB, and evaluate the impact of λ on the MTTDLs. Figure 8(b) shows the results. Both CORE and conventional recovery see a decreasing MTTDL as λ increases. From both Figures 8(a) and 8(b), CORE has a larger MTTDL than conventional recovery (by 10-100 times), since it has a higher recovery rate with less recovery bandwidth. For example, considering T =1TB, B =1Gbps and λ =0.25, the MTTDL of CORE is 26 times that of conventional recovery.

4 Implementation

We complement our theoretical analysis with a prototype implementation. As a proof of concept, we implement CORE as an extension to the Hadoop Distributed File System (HDFS) [49]. We modify the source code of HDFS and its erasure code module HDFS-RAID [22]. We point out that CORE is also applicable for general large-scale distributed storage systems.

2. Recall that we assume $n = 2k$ (see Section 3.1), and hence $t < k = n - k$ and we can apply Theorem 1.



(a) Transfer rate B (b) Node failure rate λ

Fig. 8. Comparisons of MTTDLs of CORE and conventional recovery for different transfer rates and node failure rates.

4.1 Overview of HDFS-RAID

Files stored in HDFS [49] are divided into large-size blocks (e.g., 64MB by default), which we refer to as "segments" in this paper and form the basic units of HDFS read/write operations. By default, HDFS keeps three replicas for each segment to achieve data availability. To provide data availability with smaller storage overhead, HDFS-RAID is designed to convert replicas into erasure-coded data and distribute the erasure-coded data across different nodes. We call the whole conversion process the *striping* operation.

HDFS-RAID uses a distributed RAID file system (DRFS) to manage the erasure-coded data stored in HDFS. HDFS-RAID adds a new node called RaidNode, which performs the striping operation and coordinates the recovery operation. Also, HDFS-RAID provides a client-side interface called the DRFS client, which handles all read/write requests for the erasure-coded data stored in HDFS. If a lost segment is requested, then it performs degraded read to the lost segment.

The striping operation is carried out as follows. For a given (n, k), the RaidNode first downloads a group of k segments (from one of the replicas for each segment). It then encodes the k segments into n segments on a per-stripe basis (see Section 2.1). The n segments are then placed on n DataNodes. Unused replicas of the k segments will later be removed from HDFS. The RaidNode repeats the same process for another group of k segments.

4.2 Integration into HDFS-RAID

To integrate our relayer model into HDFS-RAID, we can simply deploy a relayer daemon in the RaidNode and the DRFS client for failure recovery and degraded reads, respectively. CORE is implemented on HDFS release 0.22.0 with HDFS-RAID enabled. We modify both the RaidNode and the DRFS client to support concurrent recovery. Since regenerating codes need DataNodes to generate encoded packets during recovery, we add a signal handler in each DataNode to respond to requests for encoded packets. During recovery, the RaidNode or the DRFS client notifies the surviving DataNodes about the identities of the failed nodes, and the DataNodes accordingly generate the encoded packets.

Optimizations of coding. In our current prototype, we implement RS codes [41] and IA codes [51] as candidates of erasure codes and regenerating codes, respectively. We implement them in the ErasureCode module of HDFS-RAID. To minimize the computational overhead of the encoding/decoding operations, we implement the coding schemes in C++ using the Jerasure library [36], and have the ErasureCode module execute a specific coding scheme through the Java Native Interface (note that HDFS-RAID is written in Java). For each code we implement, we add *XOR transformation* [3], which changes all encoding/decoding operations into purely XOR operations, and *XOR scheduling* [21], which reduces the number of redundant XOR operations during encoding/decoding. Both XOR transformation and XOR scheduling are available in the Jerasure library [36].

Pipelined model. The original HDFS-RAID uses a single-threaded implementation. For further speedup, we implement a *pipelined* model that leverages multi-threading to parallelize the encoding/decoding operations. Figure 9 shows the implementation of our pipelined design in CORE, assuming that a single failure is to be recovered. The RaidNode requests metadata from the NameNode (Steps 1-2) and downloads segments from the surviving nodes (Steps 3-4). Then the RaidNode reconstructs the lost data using the pipelined implementation, which is composed of three stages. First, we have an *input thread* that collects data from the surviving DataNodes. For regenerating codes, the input thread fetches the corresponding encoded packets from DataNodes. The input thread then dispatches the data via a shared circular buffer to the *worker thread*, which decodes the lost data for the failed nodes. It sends the decoded lost data to an *output thread*, which then uploads the resulting segments (Step 5).

5 Prototype Experiments

We perform experiments on CORE, using a distributed storage system testbed. A major deployment issue is that the overall performance of recovery is determined by a combination of factors including network bandwidth, disk I/Os, encoding/decoding overhead. We address the following questions: (i) Does minimizing recovery bandwidth play a key role in improving the overall recovery performance (see Section 5.1)? (ii) Can CORE preserve the performance of the normal striping operation offered by HDFS-RAID (see Section 5.2)? (iii) How much can CORE improve the performance of recovery, degraded reads, and MapReduce (see Sections 5.3-5.5)?

We conduct our experiments on an HDFS testbed with one NameNode and up to 20 DataNodes. Each node runs on a quad-core PC equipped with an Intel Core i5-2400 3.10GHz CPU, 8GB RAM, and a Seagate ST31000524AS 7200RPM 1TB SATA hard disk. All machines are equipped with a 1Gb/s Ethernet card and

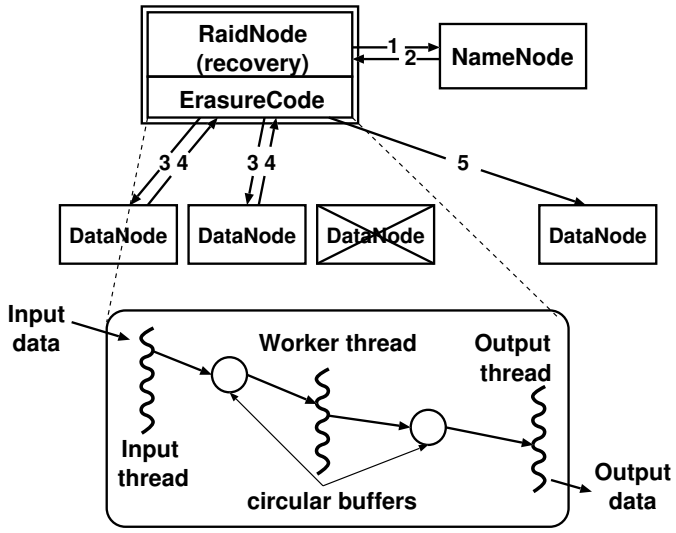


Fig. 9. Illustration of the pipelined implementation in CORE for the recovery operation, assuming that we recover a single failure. The same implementation applies to striping (in the RaidNode) and degraded reads (in the DRFS client).

interconnected over a 1Gb/s Ethernet switch. They all run Linux Ubuntu 12.04.

We compare RS codes [41], which use conventional recovery, and CORE, which builds on IA codes [51] (see Section 4.2). Both codes are implemented in C++ and compiled with g++ 4.6.3 with the -O3 option. Our microbenchmark results (see Section 5.1) are averaged over 10 runs, while the macrobenchmark results are averaged over five runs.

5.1 Microbenchmark Studies

In this subsection, we conduct microbenchmark studies on the recovery operation. We first evaluate the decoding performance versus the packet size. We then provide a breakdown analysis on different recovery steps.

Decoding performance. To evaluate the computational overhead of RS codes and CORE in recovery, we measure how fast the relayer decodes the lost data using the packets downloaded from surviving nodes. Since the decoding operations are performed over packets (see Section 2.1), we study how the packet size affects the decoding performance.

We vary the packet size from 8 bytes to 32KB. Our evaluation operates on 30 stripes of data for different sets of (n, k). To stress test the computational performance, we eliminate the impact of disk I/Os by first loading the data that is to be downloaded by the relayer for recovery into memory. We then measure the time for performing all decoding operations on the in-memory data. We compute the *decoding throughput*, defined as the size of the lost data divided by the decoding time.

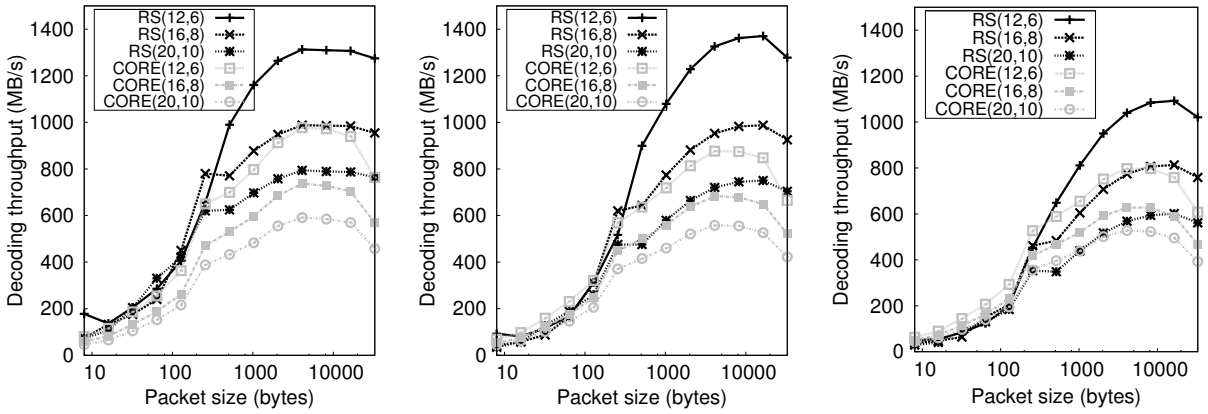
Figure 10 shows the decoding throughput for one to three failures for RS codes and CORE. Larger (n, k) implies more failures can be tolerated, but has smaller decoding throughput since the generator matrix becomes larger and the decoding overhead is higher. Note that the throughput trend versus the packet size also conforms to the results of different erasure codes in the study [36]. The throughput initially increases with the packet size, and reaches maximum when the packet size is around 4KB to 8KB. When the packet size further increases, the throughput drops because of cache misses [36].

RS codes have higher decoding throughput than CORE (which builds on IA codes). The reason is that although CORE downloads less data (i.e., fewer packets) from surviving nodes than RS codes, it decodes each lost packet from $(n-t)t$ downloaded packets, while RS codes decode each lost packet from k downloaded packets. Thus, CORE has a higher computational complexity than RS codes for the same (n, k). Nevertheless, in all cases we consider, CORE has at least 500MB/s of decoding throughput at packet size 8KB. Our following benchmark results show that the decoding performance is *not* the bottleneck in the recovery operation.

Breakdown analysis. Recall from Figure 2 that a recovery operation can be decomposed into five different steps. We now conduct a simplified analysis on the expected performance of each recovery step in RS codes and CORE. Our goal is to identify the bottleneck, so as to justify the need of minimizing recovery bandwidth.

We fix the storage capacity of each node to be 1GB. Suppose that we recover t failed nodes with a total of tGB of data, and that $(n, k) = (20, 10)$ is used. We collect the system parameters based on the measurements on our testbed hardware, and derive the expected time for each recovery step as shown in Table 2. We elaborate our derivations as follows.

- *I/O step.* In both RS codes and CORE, each surviving node reads all its stored data. For our disk model, our measurements (using the Linux command `hdparm`) indicate that the disk read speed is 116MB/s. Suppose that all surviving nodes read data in parallel. In the I/O step, both schemes take 1GB÷116MB/s $\approx$ 8.83s.
- *Encode step.* In RS codes, surviving nodes do not perform encoding, while in CORE, surviving nodes encode their stored data. Suppose that all surviving nodes perform the encode step in parallel. Our measurements indicate that the encoding time on an i5-2400 machine is no more than 0.4 seconds for 1GB of raw data.
- *Download step.* The relayer downloads data from other surviving nodes via its 1Gb/s interface, so its effective transfer rate is upper bounded by 1Gb/s (or 125MB/s). For RS codes, the relayer always downloads the same amount of original data, which is $k\times$1GB = 10GB. For CORE, we consider only the good failure patterns, which account for the majority of cases (see Section 3.4). From Theorem 1, the relayer downloads $0.1t(20-t)$GB of data (where $t < k = 10$). We can derive the (minimum) down-



(a) $t = 1$ (b) $t = 2$ (c) $t = 3$

Fig. 10. Decoding throughput of RS codes and CORE versus the packet size for different (n, k).

TABLE 2
Time comparisons for different recovery steps in RS codes and CORE in (20,10), with 1GB data per node.

time(s)	RS $t=1$	RS $t=2$	RS $t=3$	CORE $t=1$	CORE $t=2$	CORE $t=3$
I/O	8.83	8.83	8.83	8.83	8.83	8.83
Encode	0	0	0	0.12	0.23	0.35
Download	81.92	81.92	81.92	15.56	29.49	41.78
Decode	1.30	2.75	5.17	1.75	3.69	5.87
Upload	8.19	16.38	24.58	8.19	16.38	24.58

load times for RS codes and CORE accordingly. In reality, the effective transfer rate is lower than 1Gb/s and the download times will be higher.

- *Decode step.* We fix the packet size at 8KB, in which both RS codes and CORE can achieve high decoding throughput according to our previous experiments. The decoding throughput values of RS codes are 594-789MB/s, while those of CORE are 523-585MB/s. We derive the decoding times by dividing tGB by the decoding throughput for t failures.
- *Upload step.* The relayer uploads tGB of decoded data via its 1Gb/s interface. We derive the upload times as in the download step.

From our derivations, we see that the download step uses the most time among all operations. Since we can pipeline the download, decode, and upload steps in the relayer, we see that the download step is the bottleneck. This justifies the need of minimizing recovery bandwidth, which we define as the amount of data transferred in the download step.

5.2 Striping

We now evaluate the striping operation that is originally provided by HDFS-RAID when encoding replicas with RS codes and IA codes (used by CORE). We also compare our pipelined implementation with the original single-threaded implementation in HDFS-RAID. Our goal is to show that CORE, when using IA codes, maintains the striping performance when compared to RS codes.

For a given (n, k), we configure our HDFS testbed with n DataNodes, one of which also deploys the RaidNode. We prepare a kGB of original data as our input. By our observation, the input size is large enough to give a steady throughput. HDFS first stores the file with the default 3-replication scheme. Then the RaidNode stripes the data into encoded data using either RS codes or IA codes. The encoded data is stored in n DataNodes. For load balancing, we rotate node identities when we place the segments so that the data and parity segments are evenly distributed across DataNodes. We fix the packet size at 8KB. We set the segment size at 64MB, which is default setting in HDFS, but for some (n, k), we alter the segment size slightly to make it a multiple of the strip size (which is $(n-k)\times$8KB) for IA codes. We measure the *striping throughput* as the original size of data divided by the total time for the entire striping operation.

Figure 11 shows the striping throughput results. By parallelizing the data transfer and encoding/decoding steps, our pipelined implementation improves the striping throughput by around 50% over the original single-threaded implementation in HDFS-RAID. We see that IA codes have smaller striping throughput than RS codes in both implementations. In single-threaded implementation, IA codes have higher encoding/decoding overhead and hence show worse performance. In pipelined implementation, IA codes have strip size $r = n - k$ and contain more packets per stripe than RS codes with strip size $r = 1$. Our pipelined implementation will not start the encoding thread until the RaidNode downloads the first stripe of packets for each group of k segments (see Section 4.1). Thus, RS codes benefit more from parallelization. However, the throughput drop in IA codes is small, by at most 6.1% only in our pipelined implementation.

5.3 Recovery

We evaluate the performance of recovery. We first stripe encoded data across DataNodes as in Section 5.2. Then we manually delete all segments stored on t DataNodes

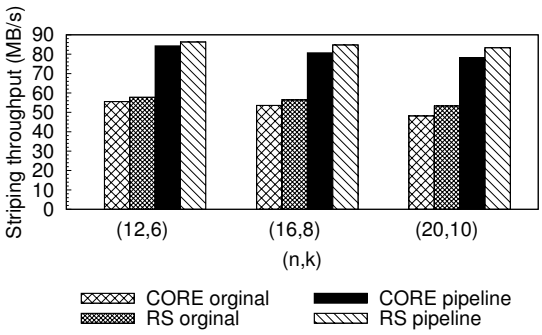


Fig. 11. Striping throughput.

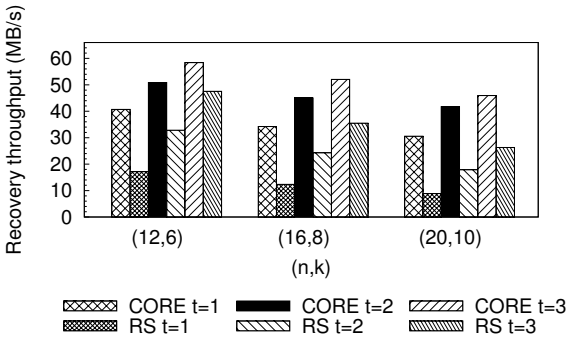


Fig. 12. Recovery throughput.

to mimic t failures, where $t = 1, 2, 3$. Since we rotate node identities when we stripe data, the lost segments of the t failed DataNodes include both data and parity segments. The RaidNode recovers the failures and uploads reconstructed segments to new DataNodes (same as the failed DataNodes in our evaluation). Here, we deploy the RaidNode in one of the new DataNodes. We measure the *recovery throughput* as the total size of lost segments divided by the total recovery time.

Figure 12 shows the recovery throughput results. Both RS codes and CORE see higher throughput for larger t as more lost segments are recovered. Overall, CORE shows significantly higher throughput than RS codes. The throughput gain is the highest in (20,10). For example, for single failures, the gain is 3.45×; for concurrent failures, the gains are 2.33× and 1.75× for $t = 2$ and $t = 3$, respectively.

Our experimental results are fairly consistent with our analytical results in Section 3.4. For example, in (20,10), the ratio of the recovery bandwidth of CORE to that of erasure codes for $t = 2$ and $t = 3$ are 0.36 and 0.51, respectively (see Figure 6(a)). These results translate to the recovery throughput gains of CORE at 2.78× and 1.96×, respectively. Our experimental results show slightly less gains, mainly due to disk I/O and encoding/decoding overheads that are not captured in the recovery bandwidth.

5.4 Degraded Reads

We further evaluate the performance of degraded reads in the presence of transient failures. The evaluation

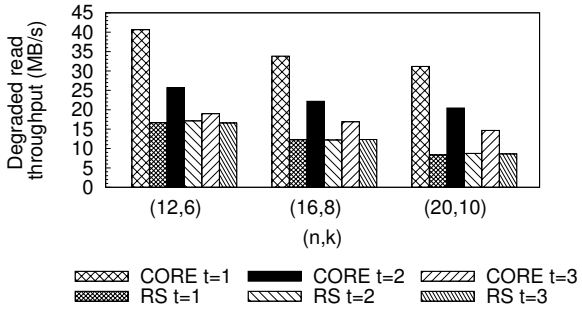


Fig. 13. Degraded read throughput.

setting is the same as that of the recovery operation described in Section 5.3, except that the degraded read operation is now performed by the DRFS client. Suppose that t nodes fail, where $t = 1, 2, 3$. We have the DRFS client request a lost segment on one of the failed DataNodes. The lost segment will be reconstructed from the data of other surviving DataNodes. Here, we deploy the DRFS client in one of the failed DataNodes. We measure the *degraded read throughput*, defined as the amount of data being requested divided by the response time.

Figure 13 shows the degraded read throughput results. RS codes maintain almost the same throughput for each (n, k), as they always download k segments for reconstruction. Overall, CORE shows a throughput gain in degraded reads. For example, if we consider the (20,10) code, CORE shows degraded throughput gain of 3.75×, 2.34× and 1.70× for $t = 1$, $t = 2$ and $t = 3$, respectively.

Note that CORE is optimized for reconstructing t lost segments on t failures. If only one lost segment is reconstructed while $t > 1$, it is possible to use even less recovery bandwidth. Nevertheless, our results still show the improvements of CORE over the conventional one.

5.5 Runtime of MapReduce with Node Failures

MapReduce [9] is an important data-processing framework running on top of HDFS. Here, we conduct a preliminary evaluation on how CORE affects the performance of a MapReduce job with node failures.

We run a classical WordCount job using MapReduce to count the words in a document collection. The WordCount job runs a number of tasks of two types: a *map* task reads a segment from HDFS and emits each word to a *reduce* task, and the reduce task aggregates the results of multiple map tasks. With node failures, some map tasks may perform degraded reads to the unavailable segments.

We consider the same evaluation settings as in Section 5.4. Here, we focus on (20,10). We run a WordCount job on 10GB of plain text data obtained from the Gutenberg website [19]. Using CORE or RS codes, we stripe the encoded segments across DataNodes, disable t nodes to simulate a t-node failure, and then run the WordCount job on the encoded data. We also consider the baseline MapReduce job when there is no failure. We use the

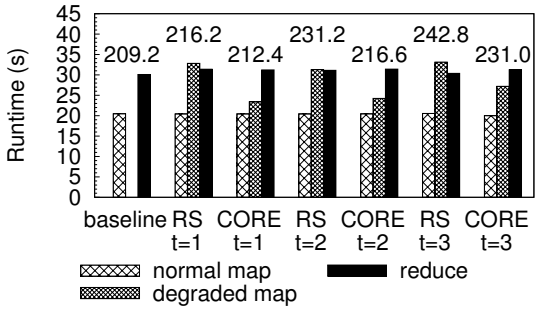


Fig. 14. Runtimes of different types of MapReduce tasks. The overall runtime of the MapReduce job (in seconds) in each setting is also shown above the bars.

default MapReduce scheduler to schedule tasks across DataNodes.

We measure the runtime performance of different types of MapReduce tasks: (i) the average runtime of a normal map task that runs on the available data of normal nodes, (ii) the average runtime a degraded map task that runs on the unavailable data of failed nodes, and (iii) the average runtime of a reduce task. We define the runtime of a map/reduce task as the duration from when the task initialization function is called until when the task completion function is called. The runtime of a map task captures the time of reading the segments to be processed; if a map task is degraded and accesses unavailable data, it issues a degraded read. On the other hand, the runtime of a reduce task only captures the processing time for the intermediate results of the map tasks (i.e., the time of transmitting data from the map tasks to reduce tasks is excluded). In addition to the task runtimes, we also measure the overall runtime of the WordCount job, defined as the duration from when it starts until when it completes.

Figure 14 shows the runtimes of different MapReduce components. For the normal map tasks and the reduce tasks, their runtimes are almost identical to the baseline, meaning that CORE does not have adverse effects to such tasks. The degraded map task incurs a longer time than the baseline due to degraded reads. Nevertheless, CORE outperforms RS codes in this item. For t =1, 2 and 3, CORE takes 29%, 22% and 18% less time than RS codes to run a degraded map task. The results also conform to our theoretical findings. The extra runtime of the degraded map task over the normal map task is mainly due to the degraded read request. Consider $t = 1$. For RS codes, the extra runtime is 12.4s, while for CORE, the extra runtime is 2.96s (or 80% less). This is consistent with our analysis results in Section 3.4.

CORE also improves the overall runtime of the WordCount job, although the improvement is less significant due to other overheads. On the other hand, we expect that the improvement of CORE becomes more significant in a large-scale distributed setting where network bandwidth is limited. We argue that the MapReduce evaluation here is preliminary. We plan to consider more workloads and testbed environments in future work.

6 Related Work

We review related work on the recovery problem for erasure codes and regenerating codes.

Minimizing I/Os. Several studies focus on minimizing I/Os required for recovering a single failure in erasure codes. Their approaches mainly focus on a disk array system where the disk access is the bottleneck. The authors of [53], [55] propose optimal single failure recovery for RAID-6 codes. Khan *et al.* [30] show that finding the optimal recovery solution for arbitrary erasure codes is NP-hard. Note that the performance gains of the above solutions over the conventional recovery are generally less than 30%, while regenerating codes achieve a much higher gain in single failure recovery (see Section 5).

The authors of [13], [26], [34], [42][3] have proposed local recovery codes that reduce bandwidth and I/O when recovering lost data. They evaluate the codes atop a cloud storage system simulator (e.g., in [34]), Azure Storage (e.g., in [26]) and HDFS (e.g., in [13], [42]). It is worth noting that the local recovery codes are non-MDS codes with additional parities added to storage, so as to trade for better recovery performance. All these studies focus on optimizing single failure recovery. Our work differs from them in several aspects: (i) we consider optimal minimum storage regenerating codes that are MDS codes, (ii) we consider recovering both single and concurrent failures, (iii) we implement and perform testbed experiments on regenerating codes that require storage nodes to perform encoding operations.

Minimizing the bandwidth. Regenerating codes [11] minimize the bandwidth for single failure recovery in a distributed storage system. There have been many theoretical studies on constructing regenerating codes (e.g., [4], [11], [35], [37], [38], [44], [51], [52]). Most regenerating code constructions require surviving nodes to encode stored data during recovery. Some regenerating code constructions (e.g., [23], [40], [45], [54]) eliminate such encoding operations during recovery so as to minimize both I/Os and bandwidth, but they make different trade-offs (see discussion in [24]). Implementation studies of regenerating codes have recently received attention from the research community, such as [12], [23], [27], [28]. Note that the studies [12], [27], [28] do not integrate regenerating codes into a real storage system, while NCCloud [23] implements a storage prototype based on non-systematic regenerating codes.

Cooperative recovery. Several theoretical studies (e.g., [25], [29], [47], [48]) address concurrent failure recovery based on regenerating codes, and they focus on recovery of lost data on new nodes. They all consider

3. Although the proposed scheme of [13] is also called CORE, it refers to Cross Object Redundancy and builds on local recovery codes, which have very different constructions from regenerating codes considered by our work.

a *cooperative model*, in which the new nodes exchange among themselves their data being read from surviving nodes during recovery. The authors of [25], [29] prove that the cooperative model achieves the same optimal recovery bandwidth as ours, but they do not provide explicit constructions of regenerating codes that achieve the optimal point. The authors of [47], [48] provide such explicit implementations, but they focus on limited parameters and the resulting implementations do not provide any bandwidth saving over erasure codes. A drawback of the cooperative model is that new nodes need to perform decoding operations and exchange decoded data among themselves, and its implementation complexities are unknown. In contrast, CORE performs all decoding operations in the relayer and is more easily implemented.

7 Discussion

In this section, we discuss some open issues that are not covered in this paper.

High redundancy of CORE. In this paper, we consider the MSR codes with fairly high redundancy (i.e., double redundancy), due to the requirements imposed by the underlying constructions of optimal exact regenerating codes. It is shown in [44] that all (n, k) linear MSR codes with exact recovery must satisfy the condition $n \geq 2k - 2$. Other (n, k) codes may be constructed via the non-systematic, functional regenerating codes [11], which are suited to archival data [23]. How to extend CORE for functional regenerating codes remains an open issue.

Concurrent recovery of non-MDS codes. We consider the concurrent recovery problem of MSR codes, which achieve the minimum storage efficiency as in MDS codes (see Section 2.1). One may consider the non-MDS codes, which incur higher storage overhead but achieve better single failure recovery performance (e.g. MBR codes [38] and local recovery codes [13], [26], [34], [42]). An open issue is how to extend these non-MDS codes to support efficient concurrent recovery.

Wide-area storage systems. We currently implement CORE atop HDFS. We plan to explore the implementation of CORE in wide-area storage systems (e.g., [2], [7], [8], [31]), where network bandwidth is limited and the benefits of regenerating codes should become more prominent. Also, one side benefit of CORE is that we can delay recovery until the number of failed nodes reaches some threshold so as to we avoid recovering transient failures that are commonly found in wide-area networks [2], [6], [33].

8 Conclusions

We address the recovery problem in a distributed storage system in the presence of single and concurrent failures, from both theoretical and applied perspectives. We explore the use of regenerating codes (or network coding) to provide fault-tolerant storage and minimize the bandwidth of data transfer during recovery. We propose a system CORE, which generalizes existing optimal single-failure-based regenerating codes to support the recoveries of both single and concurrent failures. We theoretically show that CORE minimizes the recovery bandwidth in most concurrent failure patterns. We further prototype CORE as a layer atop Hadoop HDFS, and show via testbed experiments that we can speed up both recovery and degraded read operations. The source code of our CORE prototype is available for download at: **http://ansrlab.cse.cuhk.edu.hk/software/core**.

Acknowledgments

This work is supported by grants AoE/E-02/08 and ECS CUHK419212 from the University Grants Committee of Hong Kong.

References

[1] R. Ahlswede, N. Cai, S. Li, and R. Yeung. Network Information Flow. *IEEE Trans. on Information Theory*, 46(4):1204–1216, Jul 2000.
[2] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of USENIX NSDI*, Oct 2004.
[3] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based Erasure-Resilient Coding Scheme. Technical report, The International Computer Science Institute, Berkeley, CA, Aug 1995.
[4] V. R. Cadambe, C. Huang, and J. Li. Permutation Code: Optimal Exact-Repair of a Single Failed Node in MDS Code Based Distributed Storage Systems. In *Proc. of IEEE ISIT*, 2011.
[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.
[6] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proc. of USENIX NSDI*, May 2006.
[7] Cleversafe. http://www.cleversafe.com.
[8] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, Dec 2001.
[9] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proc. of USENIX OSDI*, Dec 2004.
[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proc. of ACM SOSP*, 2007.
[11] 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, Sep 2010.
[12] A. Duminuco and E. Biersack. A Practical Study of Regenerating Codes for Peer-to-Peer Backup Systems. In *Proc. of IEEE ICDCS*. IEEE, Jun 2009.
[13] K. Esmaili, P. Lluis, and A. Datta. CORE: Cross-Object Redundancy for Efficient Data Repair in Storage Systems. In *Proc. of IEEE BigData*, 2013.
[14] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.
[15] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.
[16] K. Greenan. *Reliability and Power-Efficiency in Erasure-Coded Storage Systems*. PhD thesis, University of California, Santa Cruz, 2009.
[17] K. M. Greenan, E. L. Miller, and T. J. E. Schwarz. Optimizing Galois Field Arithmetic for Diverse Processor Architectures and Applications. In *Proc. of IEEE MASCOTS*, 2008.
[18] K. M. Greenan, J. S. Plank, and J. J. Wylie. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *Proc. of USENIX HotStorage*, 2010.

[19] Gutenberg. http://www.gutenberg.org/.

[20] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proc. of USENIX NSDI*, May 2005.

[21] J. Hafner, V. Deenadhayalan, K. Rao, and J. Tomlin. Matrix Methods for Lost Data Reconstruction in Erasure Codes. In *Proc. of USENIX FAST*, Dec 2005.

[22] HDFS-RAID. http://wiki.apache.org/hadoop/HDFS-RAID.

[23] Y. Hu, H. Chen, P. Lee, and Y. Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *Proc. of USENIX FAST*, Feb 2012.

[24] Y. Hu, P. P. C. Lee, and K. W. Shum. Analysis and Construction of Functional Regenerating Codes with Uncoded Repair for Distributed Storage Systems. In *Proc. of IEEE INFOCOM*, Apr 2013.

[25] Y. Hu, Y. Xu, X. Wang, C. Zhan, and P. Li. Cooperative Recovery of Distributed Storage Systems from Multiple Losses with Network Coding. *IEEE Journal on Selected Areas in Communications (JSAC)*, 28(2):268–276, Feb 2010.

[26] 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*, Jun 2012.

[27] Z. Huang, E. Biersack, and Y. Peng. Reducing Repair Traffic in P2P Backup Systems: Exact Regenerating Codes on Hierarchical Codes. *ACM Trans. on Storage*, 7(3):10, Oct 2011.

[28] S. Jiekak, A.-M. Kermarrec, N. L. Scouarnec, G. Straub, and A. Van Kempen. Regenerating Codes: A System Perspective. *ACM SIGOPS Operating Systems Review*, 47(2):23–32, Jul 2013.

[29] A. Kermarrec, N. Le Scouarnec, and G. Straub. Repairing Multiple Failures with Coordinated and Adaptive Regenerating Codes. In *Proc. of NetCod*, Jun 2011.

[30] 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*, Feb 2012.

[31] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proc. of ACM ASPLOS-IX*, Nov 2000.

[32] R. Li, J. Lin, and P. P. C. Lee. CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems. In *Proc. of IEEE MSST*, May 2013.

[33] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proc. of USENIX NSDI*, May 2006.

[34] D. Papailiopoulos, J. Luo, A. Dimakis, C. Huang, and J. Li. Simple Regenerating Codes: Network Coding for Cloud Storage. In *Proc. of IEEE INFOCOM*, Mar 2012.

[35] D. S. Papailiopoulos, A. G. Dimakis, and V. R. Cadambe. Repair Optimal Erasure Codes through Hadamard Designs. In *Proc. of Allerton Conf.*, 2011.

[36] 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*, Feb 2009.

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

[38] K. Rashmi, N. Shah, P. Kumar, and K. Ramchandran. Explicit Construction of Optimal Exact Regenerating Codes for Distributed Storage. In *Proc. of Allerton Conf.*, Sep 2009.

[39] K. 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 WSarehouse Cluster. In *Proc. of USENIX HotStorage*, 2013.

[40] K. Rashmi, N. B. Shah, and K. Ramchandran. A Piggybacking Design Framework for Read-and-Download-Efficient Distributed Storage Codes. In *Proc. of IEEE ISIT*, 2013.

[41] I. Reed and G. Solomon. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, Jun 1960.

[42] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proc. of VLDB Endowment*, 2013.

[43] B. Schroeder and G. A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *Proc. of USENIX FAST*, Feb 2007.

[44] N. Shah, K. Rashmi, P. Kumar, and K. Ramchandran. Interference Alignment in Regenerating Codes for Distributed Storage: Necessity and Code Constructions. *IEEE Trans. on Information Theory*, 58(99):2134 – 2158, Apr 2012.

[45] N. B. Shah. On Minimizing Data-Read and Download for Storage-Node Recovery. *IEEE Communications Letters*, 17(5):964–967, 2013.

[46] N. B. Shah, K. Rashmi, P. V. Kumar, and K. Ramchandran. Explicit Codes Minimizing Repair Bandwidth for Distributed Storage. In *IEEE Information Theory Workshop*, 2010.

[47] K. Shum. Cooperative Regenerating Codes for Distributed Storage Systems. In *Proc. of IEEE ICC*, Jun 2011.

[48] K. Shum and Y. Hu. Exact Minimum-Repair-Bandwidth Cooperative Regenerating Codes for Distributed Storage Systems. In *Proc. of IEEE ISIT*, Jul 2011.

[49] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[50] M. Silberstein, L. Ganesh, Y. Wang, L. Alvisi, and M. Dahlin. Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-Coded Distributed Storage. In *Proc. of ACM SYSTOR*, 2014.

[51] C. Suh and K. Ramchandran. Exact-Repair MDS Code Construction using Interference Alignment. *IEEE Trans. on Information Theory*, 57(3):1425–1442, Mar 2011.

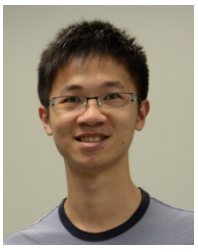
[52] I. Tamo, Z. Wang, and J. Bruck. Zigzag Codes: MDS Array Codes with Optimal Rebuilding. *IEEE Trans. on Information Theory*, 59(3):1597–1616, 2013.

[53] Z. Wang, A. Dimakis, and J. Bruck. Rebuilding for Array Codes in Distributed Storage Systems. In *IEEE GLOBECOM Workshops*, Dec 2010.

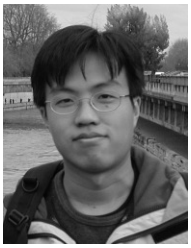
[54] Z. Wang, I. Tamo, and J. Bruck. Long MDS Codes for Optimal Repair Bandwidth. In *Proc. of IEEE ISIT*, 2012.

[55] 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, Oct 2011.

Runhui Li received his B.Eng. degree in Computer Science and Technology from University of Science and Technology of China in 2011. He is currently working toward the Ph.D. degree in Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in distributed systems and data storage.



Jian Lin received his B.Eng. degree in Mathematics and Information engineering and M.Phil. degree 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.