# Relieving Both Storage and Recovery Burdens in Big Data Clusters with R-STAIR Codes

Mingqiang Li, Runhui Li, and Patrick P. C. Lee

***Abstract*—Enterprise storage clusters increasingly adopt erasure coding to protect stored data against transient and permanent failures. Existing erasure code designs not only introduce extra parity information in a storage-inefficient manner, but also consume substantial cross-rack recovery bandwidth. To relieve both storage and recovery burdens of erasure coding, we adapt our previously proposed STAIR codes into *recovery-oriented STAIR (R-STAIR)* codes, which achieve storage efficiency, recovery efficiency, and configuration generality against a mix of node and rack failures. We evaluate R-STAIR codes via analysis and Hadoop experiments. We show that by supporting mixed fault tolerance, R-STAIR codes can significantly reduce both storage and recovery burdens in storage clusters.**

***Index Terms*—Erasure coding, recovery, storage clusters.**

## I. Introduction

### A. Background and Motivation

Enterprise storage clusters are prone to failures [6], ranging from transient failures where data is temporarily unavailable (e.g., power loss, network disconnection, system upgrades, reboots) to permanent failures where data is lost (e.g., disk crashes, sector errors). Traditional storage clusters replicate (by default, three) data copies for availability, but the storage overhead of replication prohibits scalability due to the unprecedented growth of data volume. *Erasure coding* provides a redundancy alternative that provably achieves higher fault tolerance than replication with much less redundancy [17]. It is increasingly adopted in today's enterprise storage clusters, including those in Google [6], Azure [7], and Facebook [11], [12], [14].

To construct an erasure code, we can configure two parameters $N$ and $K$ (where $K < N$). Suppose that files are stored in a storage cluster at the granularity of fixed-size *chunks*. Then we can construct an $(N, K)$ code that encodes $K$ uncoded chunks (called *data chunks*) to form additional $N - K$ equal-size coded chunks (called *parity chunks*), such that any $K$ out of the $N$ data and parity chunks can reconstruct the original data. Practical erasure code constructions often satisfy two properties: (i) *maximum distance separable (MDS)*, meaning that the storage redundancy is minimum to achieve the required fault tolerance, and (ii) *systematic*, meaning that the $K$ uncoded chunks are kept in storage. We call the collection of the $N$ data and parity chunks a *stripe*, which will be distributed to $N$ distinct nodes (e.g., storage servers). A storage cluster stores multiple stripes, which are encoded identically and independently.

While erasure coding incurs less storage redundancy, the recovery of erasure-coded data triggers substantial traffic, since it needs to retrieve enough data and parity chunks for data reconstruction. Nevertheless, extensive studies (e.g., [4], [7], [10], [12], [14]) address how to improve recovery performance in erasure-coded storage, while preserving storage efficiency.

Despite the wide adoption of erasure coding, we argue that state-of-the-art erasure codes still cannot efficiently handle the *mixed* failure nature of storage clusters, in which failures can manifest at both node and rack levels. Most existing erasure codes, including the recent recovery-optimized codes [4], [7], [10], [12], [14], protect against only one level of failures. To provide fault tolerance at both node and rack levels, a commonly used approach is to place the data and parity chunks across distinct nodes that reside in distinct racks [6], [7], [9], [10], [12], [14]. However, such *flat* chunk placement causes failure recovery to always retrieve data and parity chunks across racks and hence consume significant cross-rack bandwidth. In today's storage clusters, cross-rack bandwidth is often oversubscribed, meaning that the cross-rack bandwidth available for each node in the worst case is only a fraction of the inner-rack bandwidth (e.g., 5-20× lower) [1], [2], [16].

### B. Contributions

Our primary objective is to construct erasure codes that provide *mixed fault tolerance* for storage clusters, in which failures can manifest at more than one level. Our previously proposed STAIR codes [8] provide a starting point, by addressing fault tolerance against a mix of disk and sector failures in monolithic disk arrays. STAIR codes only focus on storage efficiency. We show that a simple extension to STAIR codes can also achieve recovery efficiency in storage clusters. We refer to our extended STAIR codes as *recovery-oriented STAIR (R-STAIR)* codes, which achieve three specific design goals:

- **Storage efficiency:** R-STAIR codes reduce storage redundancy by constructing parity chunks that specifically tolerate a mix of node and rack failures;
- **Recovery efficiency:** R-STAIR codes improve recovery performance by maintaining rack-local parity chunks, so as to eliminate cross-rack traffic for commonly found single-node recovery [7], [12], [14]; and
- **Configuration generality:** The parameters of R-STAIR codes can be flexibly configured with a general construction as in STAIR codes [8], so as to address complex and diverse failure patterns in practice.

Mingqiang Li is now with Hong Kong Advanced Technology Center, Ecosystem & Cloud Service Group, Lenovo Group Ltd. This work was done when he was with the Chinese University of Hong Kong.

Runhui Li and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong.

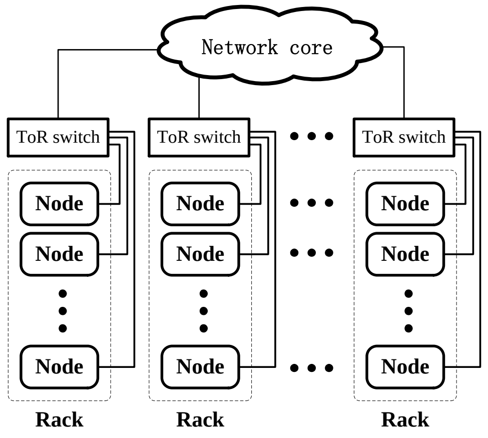


Fig. 1. Typical topology of a storage cluster.

The main contribution of this article is to demonstrate how we adapt STAIR codes [8] into storage clusters. We implement and deploy R-STAIR codes in a storage cluster testbed. We also present evaluation results for the storage and performance properties of R-STAIR codes using analysis and Hadoop experiments. We hope that our work motivates future research in providing mixed fault tolerance for data center storage. The source code of our implementation of R-STAIR codes is available online[1].

## II. Setting

We consider a simplified two-tier topology of a storage cluster as shown in Figure 1. The storage cluster is composed of multiple racks, each of which contains multiple nodes for storage. Nodes within the same rack are connected by a *top-of-rack (ToR)* switch, and the ToR switches are connected over a *network core*. In a storage cluster, inner-rack traffic traverses only ToR switches, while cross-rack traffic traverses both the ToR switches and the network core. Since cross-rack bandwidth is often oversubscribed [1], [2], [16], we assume that it is the performance bottleneck.

Failures (either transient or permanent) in a storage cluster may occur at different levels, including sectors, disks, nodes, and racks [6], [9]. This work targets two levels of failures: nodes and racks. We use a node failure to collectively include any failure of the underlying sectors or disks in a node. We consider two extremes of failure scenarios: (i) the most common *single-node failure scenario* [6], [7], [10]–[12], [14] and (ii) the *worst-case failure scenario* with simultaneous node and rack failures (i.e., the fault-tolerance limit beyond which data loss happens). In the worst-case failure scenario, we further define two rack-level failure types: (i) a *partial-rack failure*, which refers to one node failure or a burst of node failures [6] that occur in a rack, and (ii) a *full-rack failure*, which means that an entire rack fails. We explore erasure codes that not only tolerate a mix of partial-rack and full-rack failures in the worst-case failure scenario, but also mitigate the recovery overhead for the most common single-node failure scenario.

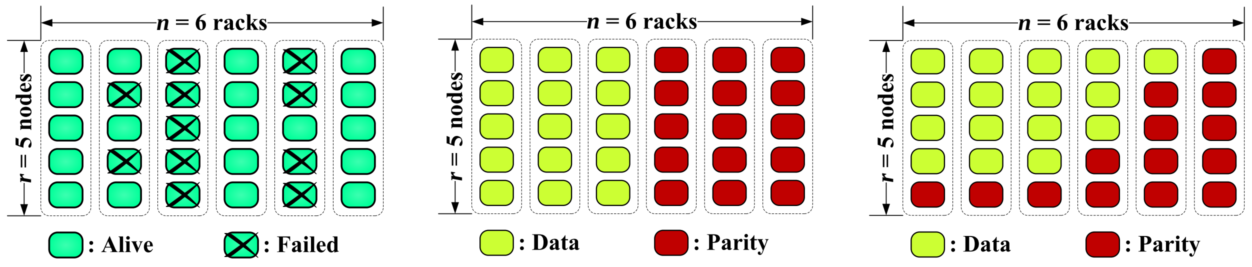
[1]http://ansrlab.cse.cuhk.edu.hk/software/rstair

Recovery for a single-node failure is generally expensive in erasure coding. For any $(N, K)$ MDS code, the conventional approach of recovering each chunk in the failed node is to retrieve any $K$ data or parity chunks of the same stripe from other non-failed nodes. Thus, the amount of *recovery traffic* (i.e., the data transferred for recovery) is $K$ times the size of recovered data. New erasure code constructions reduce the amount of recovery traffic through different techniques, such as sub-packetization (e.g., [4], [10], [12]) or local repairability (e.g., [7], [14]). For example, minimum-storage regenerating codes [4], [10] minimize the amount of recovery traffic by allowing non-failed nodes to send linear combinations of sub-packets, while ensuring the MDS property (i.e., the storage redundancy is minimized). However, to provide fault tolerance at both node and rack levels, existing erasure codes often place chunks in a flat manner (see Section I-A), such that each rack holds no more than one chunk of each stripe. As a result, their single-node failure recovery schemes still consume substantial cross-rack bandwidth.

## III. R-STAIR Codes

R-STAIR codes achieve both storage and recovery efficiencies by taking into account a mix of node and rack failures in code construction. To motivate, we compare R-STAIR codes with the classical Reed-Solomon (RS) codes [13], which are MDS codes (see Section I-A) and also assume flat chunk placement. Consider an example in which we lay out data and parity chunks, one chunk per node, over six racks with five nodes each. Suppose that our objective is to tolerate the worst-case mixed failure scenario in Figure 2(a), which contains one full-rack failure and two additional partial-rack failures with two and four failed nodes. The conventional approach of deploying RS codes is to place the data and parity chunks across different racks [6], [9]. As shown in Figure 2(b), we use RS codes to encode three data chunks to three parity chunks in each row, and distribute them across different racks. In essence, we use three entire racks to store parity chunks.

On the other hand, R-STAIR codes construct and lay out parity chunks in a different way as shown in Figure 2(c) (see the construction method in Section III-C). We place parity chunks in an entire rack to tolerate one full-rack failure, and additionally place two and four parity chunks in two other racks to tolerate *any* two partial-rack failures. Furthermore, we place one parity chunk in each of the three remaining racks. This parity layout gives R-STAIR codes two benefits. First, to recover *any* single-node failure, R-STAIR codes can retrieve the rack-local parity chunk and other non-failed data chunks from within the same rack, thereby eliminating any cross-rack traffic as opposed to RS codes. Second, while we introduce an additional parity chunk per rack for rack-local recovery, we offset the overhead by dedicating both nodes and racks to tolerating a mix of node and rack failures. In our example, RS codes incur a storage overhead of $2\times$ (see Figure 2(b)), while R-STAIR codes reduce the overhead to $1.875\times$ (see Figure 2(c)). Note that R-STAIR codes can configure different parameters to further reduce the storage overhead (see Section V-A).



(a) Worst-case failure scenario (b) RS codes (c) R-STAIR codes

Fig. 2. Example of how RS and R-STAIR codes tolerate the worst-case failure scenario.

## A. Configuration Parameters

R-STAIR codes are configured by the following six parameters, which specify how to construct and lay out a stripe of data and parity chunks:

- $n$: number of racks;
- $r$: number of nodes per rack;
- $m$: number of tolerable full-rack failures;
- $m'$: number of tolerable partial-rack failures;
- $\mathbf{e}$: a failure coverage vector $(e_0, e_1, \cdots, e_{m'-1})$ that specifies the numbers of tolerable node failures in the $m'$ tolerable partial-rack failures (where $e_0 \le e_1 \le \cdots \le e_{m'-1}$); and
- $l$: number of node failures that can be locally recovered within a rack.

As an example, the R-STAIR code in Figure 2(c) is specified by $n = 6$, $r = 5$, $m = 1$, $m' = 2$, $\mathbf{e} = (2, 4)$, and $l = 1$.

The parameters $n$, $r$, $m$, and $m'$ specify the trade-off between fault tolerance and storage efficiency. In particular, increasing $n$ and $r$ (for given $m$ and $m'$) improves storage efficiency, but degrades fault tolerance. The parameter $\mathbf{e}$ specifies how we tolerate node failure bursts in a partial-rack failure [6]. For example, $\mathbf{e} = (2, 4)$ means that we can simultaneously tolerate a burst of two-node failures and a burst of four-node failures in any two partial-rack failures. In general, if we want to tolerate a burst of $b > 1$ node failures in a rack, we may set the largest element $e_{m'-1} = b$. Note that we inherit the above five parameters from STAIR codes [8]. Here, we map the device and sector failures of STAIR codes to rack and node failures of R-STAIR codes, respectively.

The parameter $l$, on the other hand, is new in R-STAIR codes. It supports rack-local recovery by placing $l$ parity chunks in every rack. Since we assume local recovery for single-node failures, we set $l = 1$, although our R-STAIR code construction supports any value of $l$. If $l = 0$, then R-STAIR codes reduce to the original STAIR codes.

## B. Types of Parity Chunks

R-STAIR codes build on three types of parity chunks, where the parity layout is shown in Figure 3. First, there are $m$ racks of *horizontal parity chunks*, each of which is encoded from all other chunks in the same row. Second, there are $l$ rows of *vertical parity chunks*, each of which is encoded from all other chunks in the same rack. Third, there are $m'$ racks that respectively have $e_0 - l, e_1 - l, \cdots, e_{m'-1} - l$ *global parity chunks*, each of which is generated from all data chunks of the stripe. Without loss of generality, we arrange the parity chunks in a “stair” layout as shown in Figure 3. Let $D_{*,*}$, $H_{*,*}$, $V_{*,*}$, and $G_{*,*}$ denote a data chunk, a horizontal parity chunk, a vertical parity chunk, and a global parity chunk, respectively, where the two subscripts represent the row and rack indices from 0 to $r - 1$ and from 0 to $n - 1$, respectively. Also, let $X_{*,*}$ denote a chunk that can serve as either a horizontal parity chunk or a vertical parity chunk.

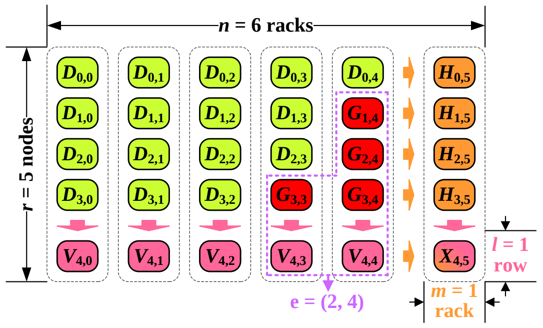


Fig. 3. Layout of a stripe of a R-STAIR code, with $n = 6$, $r = 5$, $m = 1$, $m' = 2$, $\mathbf{e} = (2, 4)$, and $l = 1$.

The three types of parity chunks serve different purposes. The horizontal parity chunks protect against $m$ rack failures as in classical RS codes [13]. The vertical parity chunks provide rack-local recovery for any $l$-node failure in the same rack. In addition, the vertical parity chunks, together with the global parity chunks, protect against partial-rack failures. For example, referring to Figure 3, the chunks $G_{1,4}$, $G_{2,4}$, $G_{3,4}$ and $V_{4,4}$ protect a burst of four-node failures in *any* partial-rack failure.

## C. Code Construction

We now describe the construction of R-STAIR codes. R-STAIR codes extend STAIR codes [8] to achieve rack-local recovery. Here, we only highlight the key steps of the construction of R-STAIR codes. We refer readers to [8] for detailed analysis and correctness proof.

R-STAIR codes operate on a *canonical stripe*, formed by augmenting a regular stripe with *virtual* parity chunks. Figure 4 illustrates how we form the canonical stripe from the regular stripe in Figure 3. The idea is to add $e_{m'-1} - l$ rows of virtual vertical parity chunks to the bottom, and add $m'$

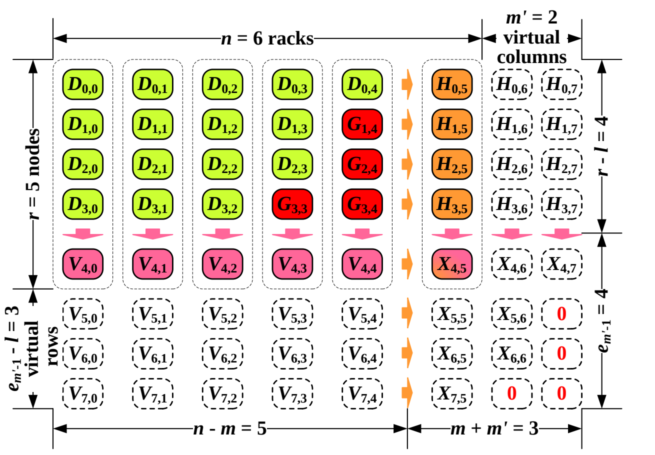

n = 6 racks
m' = 2 virtual columns
r = 5 nodes
r - l = 4
$e_{m'-1} - l = 3$ virtual rows
$e_{m'-1} = 4$
n - m = 5
m + m' = 3
$D_{0,0}$ $D_{0,1}$ $D_{0,2}$ $D_{0,3}$ $D_{0,4}$ $H_{0,5}$ $H_{0,6}$ $H_{0,7}$
$D_{1,0}$ $D_{1,1}$ $D_{1,2}$ $D_{1,3}$ $G_{1,4}$ $H_{1,5}$ $H_{1,6}$ $H_{1,7}$
$D_{2,0}$ $D_{2,1}$ $D_{2,2}$ $D_{2,3}$ $G_{2,4}$ $H_{2,5}$ $H_{2,6}$ $H_{2,7}$
$D_{3,0}$ $D_{3,1}$ $D_{3,2}$ $G_{3,3}$ $G_{3,4}$ $H_{3,5}$ $H_{3,6}$ $H_{3,7}$
$V_{4,0}$ $V_{4,1}$ $V_{4,2}$ $V_{4,3}$ $V_{4,4}$ $X_{4,5}$ $X_{4,6}$ $X_{4,7}$
$V_{5,0}$ $V_{5,1}$ $V_{5,2}$ $V_{5,3}$ $V_{5,4}$ $X_{5,5}$ $X_{5,6}$ 0
$V_{6,0}$ $V_{6,1}$ $V_{6,2}$ $V_{6,3}$ $V_{6,4}$ $X_{6,5}$ $X_{6,6}$ 0
$V_{7,0}$ $V_{7,1}$ $V_{7,2}$ $V_{7,3}$ $V_{7,4}$ $X_{7,5}$ 0 0

Fig. 4. Canonical stripe augmented from the regular stripe of the R-STAIR code depicted in Figure 3.

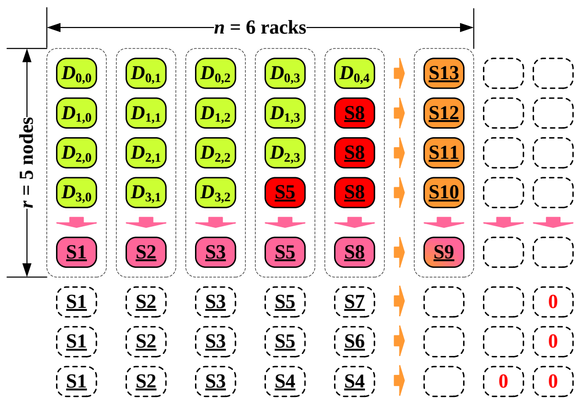
columns of virtual horizontal parity chunks to the right. Note that we create these virtual parity chunks only temporarily for encoding and decoding operations, and will discard them after the operations are completed.

R-STAIR codes perform encoding using two $(N, K)$ MDS codes, denoted by $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$, which construct parity chunks (including virtual ones) in the row and column directions, respectively. We configure $\mathcal{C}_{row}$ as an $(n + m', n - m)$ code and $\mathcal{C}_{col}$ as an $(r + e_{m'-1} - l, r - l)$ code. For example, referring to Figure 4, $\mathcal{C}_{row}$ is an $(8, 5)$ code, and $\mathcal{C}_{col}$ is an $(8, 4)$ code. We can implement $\mathcal{C}_{row}$ and $\mathcal{C}_{col}$ with any $(N, K)$ MDS codes, such as RS codes [13].

Note that the $e_{m'-1} \times (m + m')$ parity chunks at the bottom right corner of a canonical stripe (i.e., chunks $X_{*,*}$'s in Figure 4) can be encoded by either $C_{row}$ in the row direction or $C_{col}$ in the column direction, due to the homomorphic property [8]. Also, we construct the global parity chunks by setting the same number and layout of virtual parity chunks at the bottom right corner as zero values (e.g., see Figure 4).

**Encoding:** R-STAIR encoding works in an *upstairs* manner, in which we alternately encode rows of parity chunks from bottom to top via $\mathcal{C}_{row}$, and columns of parity chunks from left to right via $\mathcal{C}_{col}$. The idea is that for an $(N, K)$ MDS code, as long as there are $K$ available chunks (either data or parity chunks) of a stripe, we can generate the remaining $N - K$ chunks of the stripe. Figure 5 illustrates the sequence of steps of R-STAIR encoding based on the canonical stripe in Figure 4. We also have a similar *downstairs* encoding approach [8], which constructs parity chunks from top to bottom and right to left, and the details are omitted here.

**Decoding:** R-STAIR decoding treats any failed chunks as parity chunks and reconstructs them as in upstairs encoding. Specifically, we first logically re-arrange the rack identities so that the failed racks are mapped to the right columns (with the full-rack failures on the rightmost columns, followed by the partial-rack failures), and also re-arrange the row identities so that all failed nodes are mapped to the bottom rows. We then follow the sequence of steps of upstairs encoding to reconstruct all failed chunks. In practice, we often have much fewer failed nodes than the worst case. For a single-node failure recovery, we can locally recover the failed chunk using the vertical parity chunk located in the same rack.


n = 6 racks
r = 5 nodes
$D_{0,0}$ $D_{0,1}$ $D_{0,2}$ $D_{0,3}$ $D_{0,4}$ S13
$D_{1,0}$ $D_{1,1}$ $D_{1,2}$ $D_{1,3}$ S8 S12
$D_{2,0}$ $D_{2,1}$ $D_{2,2}$ $D_{2,3}$ S8 S11
$D_{3,0}$ $D_{3,1}$ $D_{3,2}$ S5 S8 S10
S1 S2 S3 S5 S8 S9
S1 S2 S3 S5 S7 0
S1 S2 S3 S5 S6 0
S1 S2 S3 S4 S4 0 0

**Detailed steps:**

S1: $(D_{0,0}, D_{1,0}, D_{2,0}, D_{3,0}) \overset{\mathcal{C}_{col}}{\Longrightarrow} (V_{4,0}, V_{5,0}, V_{6,0}, V_{7,0})$

S2: $(D_{0,1}, D_{1,1}, D_{2,1}, D_{3,1}) \overset{\mathcal{C}_{col}}{\Longrightarrow} (V_{4,1}, V_{5,1}, V_{6,1}, V_{7,1})$

S3: $(D_{0,2}, D_{1,2}, D_{2,2}, D_{3,2}) \overset{\mathcal{C}_{col}}{\Longrightarrow} (V_{4,2}, V_{5,2}, V_{6,2}, V_{7,2})$

S4: $(V_{7,0}, V_{7,1}, V_{7,2}, X_{7,6} = 0, X_{7,7} = 0) \overset{\mathcal{C}_{row}}{\Longrightarrow} (V_{7,3}, V_{7,4})$

S5: $(D_{0,3}, D_{1,3}, D_{2,3}, V_{7,3}) \overset{\mathcal{C}_{col}}{\Longrightarrow} (G_{3,3}, V_{4,3}, V_{5,3}, V_{6,3})$

S6: $(V_{6,0}, V_{6,1}, V_{6,2}, V_{6,3}, X_{6,7} = 0) \overset{\mathcal{C}_{row}}{\Longrightarrow} (V_{6,4})$

S7: $(V_{5,0}, V_{5,1}, V_{5,2}, V_{5,3}, X_{5,7} = 0) \overset{\mathcal{C}_{row}}{\Longrightarrow} (V_{5,4})$

S8: $(D_{0,4}, V_{5,4}, D_{6,4}, V_{7,4}) \overset{\mathcal{C}_{col}}{\Longrightarrow} (G_{1,4}, G_{2,4}, G_{3,4}, V_{4,4})$

S9: $(V_{4,0}, V_{4,1}, V_{4,2}, V_{4,3}, V_{4,4}) \overset{\mathcal{C}_{row}}{\Longrightarrow} (X_{4,5})$

S10: $(D_{3,0}, D_{3,1}, D_{3,2}, G_{3,3}, G_{3,4}) \overset{\mathcal{C}_{row}}{\Longrightarrow} (H_{3,5})$

S11: $(D_{2,0}, D_{2,1}, D_{2,2}, D_{2,3}, G_{2,4}) \overset{\mathcal{C}_{row}}{\Longrightarrow} (H_{2,5})$

S12: $(D_{1,0}, D_{1,1}, D_{1,2}, D_{1,3}, G_{1,4}) \overset{\mathcal{C}_{row}}{\Longrightarrow} (H_{1,5})$

S13: $(D_{0,0}, D_{0,1}, D_{0,2}, D_{0,3}, D_{0,4}) \overset{\mathcal{C}_{row}}{\Longrightarrow} (H_{0,5})$

Fig. 5. The sequence of steps in R-STAIR encoding, which alternately encode rows from bottom to up, and columns from left to right, in an upstairs manner.

## IV. Implementation

We implement R-STAIR codes on Facebook's Hadoop[2], which realizes both HDFS [15] and MapReduce [3]. Facebook's Hadoop also supports erasure-coded storage based on HDFS-RAID[3].

### A. Overview

We first overview HDFS, MapReduce, and HDFS-RAID. HDFS stores file data as fixed-size chunks. It is composed of a single *NameNode* and multiple *DataNodes*, in which the NameNode manages the locations and other metadata of all chunks and the DataNodes store the physical chunks. By default, HDFS replicates each chunk three times.

MapReduce [3] performs data-intensive computations on HDFS chunks. It is composed of a single *JobTracker* and multiple *TaskTrackers*, in which the JobTracker schedules computations as *jobs* with multiple *tasks*, and each task is scheduled to run on a TaskTracker. There are two types of

[2]https://github.com/facebookarchive/hadoop-20
[3]http://wiki.apache.org/hadoop/HDFS-RAID

tasks: *map* tasks, which read and process HDFS chunks, and *reduce* tasks, which collect and process outputs from map tasks.

HDFS-RAID augments HDFS with erasure coding. It adds a new *RaidNode* to manage erasure-coded data. The RaidNode launches a MapReduce job to encode replicated data chunks via erasure coding in a distributed manner. MapReduce processes erasure-coded chunks in the same way as replicated chunks on HDFS, except that when a chunk is unavailable, a map task issues *degraded reads* to retrieve other non-failed chunks of the same stripe and reconstruct the unavailable chunk. To improve degraded read performance, HDFS-RAID deploys a parallel reader to download multiple non-failed chunks simultaneously.

## B. Integration of R-STAIR Codes

We implement a C library of R-STAIR codes in the *Erasure-Code* module of HDFS-RAID, which currently implements Reed-Solomon codes. We embed the library into HDFS-RAID (written in Java) via Java Native Interface. Our experience is that the encoding/decoding operations of R-STAIR can achieve several hundred of megabytes per second, and their overheads are limited compared to network transfers.

Facebook's Hadoop currently does not support rack-local recovery. As a proof of concept, we implement a rack-local, single-node failure recovery mechanism for MapReduce. Specifically, in the original MapReduce implementation, when the JobTracker initializes a MapReduce job, it acquires the locations of all data chunks to be processed. We modify the JobTracker such that if a map task is to process an unavailable chunk on a failed DataNode, we ensure that the map task is scheduled to run on a non-failed DataNode co-located in the same rack as the failed DataNode. Thus, when the map task issues a degraded read to the unavailable chunk, it only needs to retrieve chunks of the same stripe from the same rack, without triggering cross-rack traffic.

## V. Evaluation

We compare R-STAIR codes with state-of-the-art codes in terms of storage and performance.

## A. Storage Efficiency

We demonstrate the storage efficiency of R-STAIR codes. Suppose that our goal is to tolerate $m = 1$ full-rack failure and $m' \geq 1$ partial-rack failures across $n$ racks. We measure the *storage overhead* as the total storage size normalized over the original data size. We compare R-STAIR codes with RS codes and two types of *locality-aware* codes that trade storage efficiency for recovery efficiency.

(i) *R-STAIR codes:* Recall that R-STAIR codes are parameterized by $n$, $r$, $m$, $m'$, $\mathbf{e} = (e_0, e_1, \cdots, e_{m'-1})$, and $l$. We set $l = 1$ for rack-local single-node failure recovery. We also fix $\sum_{i=0}^{m'-1} e_i = \frac{r\cdot m'}{2}$ to tolerate the failures of at most a half of the nodes in the $m'$ partial-rack failures. Note that the distribution of elements of $\mathbf{e}$ does not affect the storage overhead. Thus, the storage overhead is $\frac{r\cdot n}{r\cdot n-[r+\frac{r\cdot m'}{2}+(n-1-m')]}$. A future work is to study how to choose the most appropriate configuration parameters subject to the fault tolerance requirements.

(ii) *Reed-Solomon (RS) codes:* RS codes dedicate a rack of parity chunks to protect against a partial-rack failure. The storage overhead is $\frac{n}{n-1-m'}$. Note that Facebook's recently proposed Hitchhiker codes [12] build on the construction of RS codes, and have the same storage overhead.

(iii) *Flat local recovery (FLR) codes:* FLR codes associate local parity chunks with different subsets of nodes, and global parity chunks with all nodes in a stripe. Thus, in single-node failure recovery, FLR codes can retrieve chunks from a smaller subset of nodes. We consider Azure's implementation of FLR codes [7], and the same idea is also applicable for Facebook's one [14]. To calculate the storage overhead of FLR codes, we augment RS codes by dividing all data chunks of each stripe into two local-recovery groups and replacing one data chunk of each group with a local parity chunk. Thus, the storage overhead is $\frac{n}{n-1-m'-2}$.

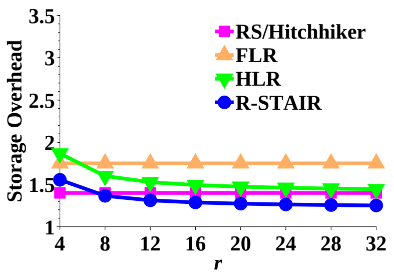
(iv) *Hierarchical local recovery (HLR) codes:* HLR codes (e.g., [5]) arrange data and parity chunks in a two-dimensional array. They place local parity chunks in each rack to allow rack locality, and place cross-rack parity chunks to tolerate rack failures. We consider HLR codes with an $r \times n$ array of chunk placement as R-STAIR codes, and augment RS codes by adding one parity chunk per rack for rack-local single-node failure recovery. The storage overhead is $\frac{r\cdot n}{r\cdot n-[r\cdot(1+m')+(n-1-m')]}$.

**Results:** We consider various configurations with default parameters $n = 14$, $r = 8$, $m = 1$, and $m' = 3$. Note that the default configuration corresponds to $(14, 10)$ RS codes used by Facebook [11], [12], [14]. Figure 6 shows the results of storage overheads. In most cases, R-STAIR codes incur the lowest storage overhead by providing mixed fault tolerance. For instance, when $n = 14$, $r = 8$, and $m' = 3$, the storage overhead of R-STAIR codes is $1.37\times$, while those of RS/Hitchhiker, FLR, and HLR codes are $1.40\times$, $1.75\times$, and $1.60\times$, respectively (or 2-28% more). To summarize, R-STAIR codes achieve higher storage savings when: (i) $r$ increases (see Figure 6(a)), as the redundancy for rack-local recovery decreases; (ii) $n$ decreases (see Figure 6(b)), as the redundancy saving is more dominant in a smaller stripe; and (iii) $m'$ increases (see Figure 6(c)), as the saving due to mixed fault tolerance becomes more dominant.

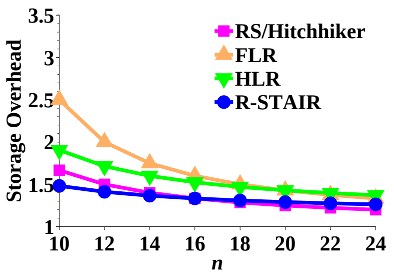
## B. Testbed Experiments

We conduct testbed experiments on Hadoop to demonstrate the recovery efficiency of R-STAIR codes. We set up a Hadoop cluster with 21 machines: one machine hosts the NameNode, RaidNode, and JobTracker, while each of the remaining 20 machines hosts a DataNode and a TaskTracker. Each machine runs Linux Ubuntu 12.04, and is equipped a 3.1GHz Intel Core i5-2400 CPU, 8GB RAM, and a Seagate ST31000524AS SATA harddisk. All machines are interconnected by a 1Gb/s Ethernet switch.

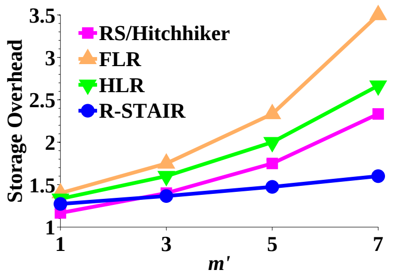
We mimic a multi-rack topology using the Linux command `tc` [1]. Specifically, to simulate the over-subscription of cross-rack bandwidth, we limit the aggregate outgoing cross-rack



(a) Varying $r$ when $n = 14$ and $m' = 3$



(b) Varying $n$ when $r = 8$ and $m' = 3$



(c) Varying $m'$ when $n = 14$ and $r = 8$

Fig. 6. Storage overheads of R-STAIR codes and existing codes.

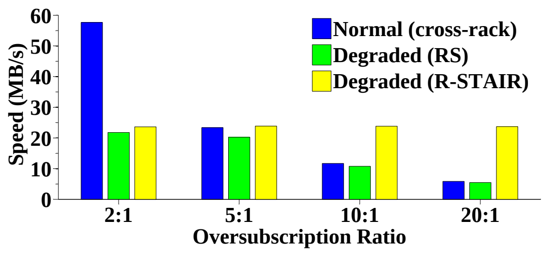


Fig. 7. Read performance of R-STAIR codes and RS codes in our Hadoop testbed cluster.

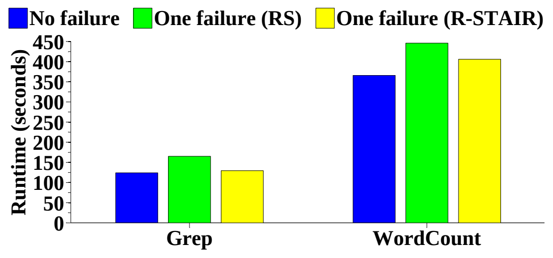
bandwidth of each node using tc, while keeping its outgoing bandwidth to each node in the same rack as 1Gb/s. For example, an over-subscription ratio $10 : 1$ means that the aggregate outgoing cross-rack bandwidth of each node is limited to 100Mb/s. Our experiments logically divide the cluster into five racks with four DataNodes each.

We compare R-STAIR codes and RS codes, where the latter is included in Facebook's Hadoop. We configure R-STAIR codes with $n = 5$, $r = 4$, $m = 1$, $m' = 1$, $\mathbf{e} = (2)$, and $l = 1$, and configure RS codes with $(5, 3)$.

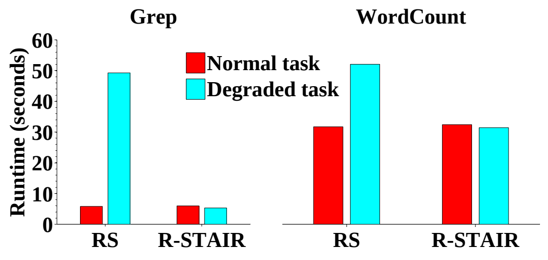
**Degraded read performance:** We first examine the degraded read performance in a single-node failure scenario. Specifically, we write a stripe into HDFS, remove one data chunk in the stripe to simulate an unavailable chunk caused by a single-node failure, and finally perform a degraded read to the unavailable chunk. We define the *degraded read speed* as the ratio of the chunk size to the degraded read time. We obtain the average results over 10 runs.

Figure 7 presents the degraded read speed results of R-STAIR codes and RS codes versus the over-subscription ratio. We also plot the normal cross-rack read speed for reference. R-STAIR codes achieve faster degraded reads than RS codes in general. As the over-subscription ratio increases, the degraded read speed of RS codes decreases from 21.79MB/s to 5.48MB/s, mainly because RS codes consume substantial cross-rack bandwidth when retrieving chunks from non-failed DataNodes in other racks, while that of R-STAIR codes is always kept around 24MB/s (up to $4.3\times$ over that of RS codes). Note that when the over-subscription ratio becomes $20 : 1$, the degraded read speed of RS codes is close to the speed of a normal cross-rack read, due to the parallel read feature of HDFS-RAID (see Section IV).

**MapReduce job runtime:** We now investigate how degraded reads of both R-STAIR codes and RS codes affect the runtime of a MapReduce job in a single-node failure. We consider two standard MapReduce jobs: *Grep* and *WordCount*. A Grep job extracts lines with a given pattern (which we use "stair") from an input file. Each map task scans a chunk of the input file and emits all matched lines to the reduce tasks, which output the matched lines. A WordCount job counts the occurrence of each word in an input file. Each map task tokenizes words in a chunk of the input file and emits each word and its occurrence count to the reduce tasks, which output the total occurrence count of each word. Both MapReduce jobs run on a 330-chunk (about 21GB) input plain text file obtained from the Gutenberg website[4].



(a) Runtime of MapReduce jobs



(b) Runtime of map tasks

Fig. 8. Runtime results of MapReduce jobs and their map tasks in our Hadoop testbed cluster.

In our evaluation, we first write an input file into HDFS, and encode the file into erasure-coded stripes with either R-STAIR codes or RS codes. We then remove all chunks stored in one DataNode to simulate a single-node failure. Finally, we run a MapReduce job to process the file data in degraded mode. We fix the over-subscription ratio of our testbed as $10 : 1$. We measure the average MapReduce job runtime over five runs.

Figure 8(a) presents the runtime results of both Grep and

[4]http://www.gutenberg.org

WordCount jobs in a single-node failure, as well as the runtime with no failure (in normal mode) for reference. We note that WordCount generally has higher runtime than Grep, as it generates more data from map tasks to reduce tasks. In a single-node failure, R-STAIR codes increase the runtime in normal mode by only $4.0\%$ for Grep and $10.9\%$ for WordCount, while RS codes increase the runtime in normal mode by $33.0\%$ for Grep and $21.9\%$ for WordCount. Compared to RS codes, R-STAIR codes save the runtime by $21.7\%$ for Grep and $9.0\%$ for WordCount.

R-STAIR codes are useful for the map tasks that issue degraded reads. To demonstrate, we further analyze the runtime of each map task in a MapReduce job. We divide map tasks into two types: (i) *normal task*, which issues a normal read to an available chunk, and (ii) *degraded task*, which issues a degraded read to an unavailable chunk due to a single-node failure.

Figure 8(b) presents the average runtime results of the two types of map tasks. We now observe a more significant difference between the map task runtimes of R-STAIR codes and RS codes. For R-STAIR codes, the runtime of a degraded task is comparable to that of a normal task; but for RS codes, the runtime a degraded task is $8.5\times$ and $1.6\times$ over that of a normal task for Grep and WordCount, respectively.

## VI. Conclusions

*Recovery-oriented STAIR (R-STAIR)* codes extend our previously proposed STAIR codes to provide mixed fault tolerance for storage clusters with three goals in mind: recovery efficiency, storage efficiency, and configuration generality. The idea of R-STAIR codes is to construct parity chunks that not only tolerate a mix of node and rack failures in a storage-efficient manner, but also achieve rack-local recovery for single-node failures. Evaluation results based on analysis and Hadoop experiments demonstrate the effectiveness of R-STAIR codes.

## Acknowledgments

This work was supported in part by grants from the University Grants Committee of Hong Kong (project numbers: AoE/E-02/08 and ECS CUHK419212) and Cisco University Research Program Fund (CG#593756) from Silicon Valley Community Foundation.

## References

[1] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *Proc. USENIX ATC*, 2014.

[2] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proc. ACM IMC*, 2010.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. USENIX OSDI*, 2004.

[4] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *IEEE Trans. on Information Theory*, 2010.

[5] K. S. Esmaili, L. Pamies-Juarez, and A. Datta. CORE: Cross-object redundancy for efficient data repair in storage systems. In *Proc. IEEE BigData*, 2013.

[6] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in globally distributed storage systems. In *Proc. USENIX OSDI*, 2010.

[7] 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. USENIX ATC*, 2012.

[8] M. Li and P. P. C. Lee. STAIR codes: A general family of erasure codes for tolerating device and sector failures in practical storage systems. In *Proc. USENIX FAST*, 2014.

[9] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar. f4: Facebook's warm BLOB storage system. In *Proc. USENIX OSDI*, 2014.

[10] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal codes for I/O, storage and network-bandwidth in distributed storage systems. In *Proc. USENIX FAST*, 2015.

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

[12] K. V. Rashmi, N. B. Shah, D. Gu, H. Kuang, D. Borthakur, and K. Ramchandran. A "hitchhiker's" guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proc. ACM SIGCOMM*, 2014.

[13] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 1960.

[14] M. Sathiamoorthy, M. Asteris, D. Papailiopoulous, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing elephants: Novel erasure codes for big data. In *PVLDB*, 2013.

[15] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop distributed file system. In *Proc. IEEE MSST*, 2010.

[16] A. Vahdat, M. Al-Fares, N. Farrington, R. N. Mysore, G. Porter, and S. Radhakrishnan. Scale-out networking in the data center. *IEEE Micro*, 2010.

[17] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proc. IPTPS*, 2002.

**Mingqiang Li** is now with Hong Kong Advanced Technology Center, Ecosystem & Cloud Service Group, Lenovo Group Ltd. His current research interests include cloud computing and storage systems. Email: `mingqiangli.cn@gmail.com`.

**Runhui Li** is now a PhD student at The Chinese University of Hong Kong. His research interests are in cloud storage. Email: `rhli@cse.cuhk.edu.hk`.

**Patrick P. C. Lee** is now an Associate Professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests are in various applied/systems topics including storage systems, distributed systems and networks, operating systems, dependability, and security. Email: `pclee@cse.cuhk.edu.hk`.