Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice

YUCHONG HU, Huazhong University of Science and Technology
XIAOLU LI, MI ZHANG, and PATRICK P. C. LEE, Chinese University of Hong Kong
XIAOYANG ZHANG, PAN ZHOU, and DAN FENG, Huazhong University of Science and Technology

Repair performance in hierarchical data centers is often bottlenecked by cross-rack network transfer. Recent theoretical results show that the cross-rack repair traffic can be minimized through repair layering, whose idea is to partition a repair operation into inner-rack and cross-rack layers. However, how repair layering should be implemented and deployed in practice remains an open issue. In this article, we address this issue by proposing a practical repair layering framework called *DoubleR*. We design two families of practical double regenerating codes (DRC), which not only minimize the cross-rack repair traffic but also have several practical properties that improve state-of-the-art regenerating codes. We implement and deploy DoubleR atop the Hadoop Distributed File System (HDFS) and show that DoubleR maintains the theoretical guarantees of DRC and improves the repair performance of regenerating codes in both node recovery and degraded read operations.

CCS Concepts: • **Information systems** → **Storage recovery strategies**; **Distributed storage**;

Additional Key Words and Phrases: Erasure coding

ACM Reference format:
Yuchong Hu, Xiaolu Li, Mi Zhang, Patrick P. C. Lee, Xiaoyang Zhang, Pan Zhou, and Dan Feng. 2017. Optimal Repair Layering for Erasure-Coded Data Centers: From Theory to Practice. *ACM Trans. Storage* 13, 4, Article 33 (November 2017), 24 pages.
https://doi.org/10.1145/3149349

1 INTRODUCTION

As data center storage expands at scale, failures are more prevalent in storage subsystems [16, 27, 45]. To maintain data availability and durability at low cost, modern data centers increasingly adopt *erasure coding* to protect data storage with a significantly low degree of redundancy while still preserving the same fault tolerance as traditional replication. At a high level, an erasure code

This work was supported in part by the National Natural Science Foundation of China (grants 61502191, 61401169, 61502190, and 61772222), Fundamental Research Funds for the Central Universities (grant 2017KFYXJJ065), the Hubei Provincial Natural Science Foundation of China (grant 2016CFB226), Key Laboratory of Information Storage System Ministry of Education of China, and Research Grants Council of Hong Kong (grants GRF 14216316 and CRF C7036-15G).
Authors' addresses: Y. Hu, X. Zhang, P. Zhou, and D. Feng, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China, 430074; emails: yuchonghu@hust.edu.cn, zhangxiaoyang1993@gmail.com, {panzhou, dfeng}@hust.edu.cn; X. Li, M. Zhang, and P. P. C. Lee (corresponding author), Department of Computer Science and Engineering, Chinese University of Hong Kong, Shatin, New Territories, Hong Kong, China; emails: {lixl, mzhang, pclee}@cse.cuhk.edu.hk.

https://doi.org/10.1145/3149349

works by transforming a set of uncoded fixed-size units, called *blocks*, into a larger set of coded blocks such that the set of uncoded blocks can be reconstructed from any subset of the same number of coded blocks. Each set of coded blocks is called a *stripe*, and a data center stores multiple stripes that are independently erasure coded. By distributing the coded blocks (or blocks in short) of each stripe across distinct storage nodes, a data center can provide fault tolerance against node failures. Field studies have shown the benefits of erasure coding in saving storage overhead in production. For example, Azure [25] and Facebook [33] reportedly reduce storage redundancy to 1.33× and 1.4×, respectively, as opposed to 3× in traditional triple replication [8, 18].

A drawback of erasure coding is the high repair cost. Repair operations are triggered when issuing degraded reads to unavailable blocks or recovering lost blocks from node crashes. In both cases, repairing each failed block in erasure-coded storage must retrieve multiple available blocks from other nodes for reconstruction. This leads to substantial *repair traffic*, defined as the amount of data transferred for repair. Facebook [39] reports that its erasure-coded data center generates a median size of 180TB of repair traffic per day, which in turn limits the bandwidth resources available for foreground jobs. In practice, bandwidth resources available for repair tasks are often throttled [25, 50] to limit their adverse impact on other application traffic. Thus, there has been an extensive literature on mitigating the repair cost (see Section 7). In particular, *regenerating codes* [15] are a special class of erasure codes that provably minimize the repair traffic, and there are many follow-up theoretical studies on regenerating codes. In addition, recent studies (e.g., Chen et al. [9], Li et al. [31], Pamies-Juarez et al. [35], and Rashmi et al. [38]) have prototyped regenerating codes and evaluated their practical performance in networked environments.

However, regenerating codes are still limited in addressing the hierarchical nature of data centers. Modern data centers organize nodes in racks and are oversubscribed to control operational costs [13]. Although full-bisection bandwidth is available within a rack, cross-rack bandwidth is constrained. Typical oversubscription ratios range from 5:1 to 20:1 [5, 6, 54] (i.e., the available cross-rack bandwidth per node is only 1/5 to 1/20 of the inner-rack bandwidth in the worst case); in some extremes, the ratio could reach 240:1 [22]. Cross-rack links are also shared by replica writes [11] or shuffle/join traffic of computing jobs [5, 26]. Note that geodistributed data centers [4, 16] also exhibit the similar hierarchical nature, as the bandwidth resources across geographical regions are limited [4] and intermittently congested [10] as opposed to within the same region.

To maximize fault tolerance, existing erasure-coded data centers often place each block of a stripe in a distinct node that resides in a distinct rack (i.e., one block per rack) [16, 25, 33, 38, 40, 44]. We call this *flat block placement*, which allows a data center to tolerate the same numbers of node failures and rack failures. However, this inevitably makes the repair of any failed block retrieve available blocks from other racks and hence incurs substantial *cross-rack* repair traffic, even though the repair traffic can be minimized by regenerating codes.

In this article, we propose *DoubleR*, a repair framework that is designed to minimize the cross-rack repair traffic for hierarchical data centers. DoubleR advocates a concept called *repair layering*, which splits a repair operation into inner-rack and cross-rack layers and trades (abundant) inner-rack bandwidth for (constrained) cross-rack bandwidth. Specifically, DoubleR opts for *hierarchical block placement*, which places multiple blocks of a stripe per rack, to minimize the cross-rack repair traffic at the expense of reducing rack-level fault tolerance. To repair a failed block, one selected node in each rack can perform partial repair operations internally using the available blocks from the same rack. It then sends partially repaired results across racks to a destination node, which combines the partially repaired results from multiple racks to reconstruct the failed block. Through repair layering, it is theoretically proven that the cross-rack repair traffic can be minimized through a new class of regenerating codes called *double regenerating codes* (DRC) [24]. We augment the theoretical results in Hu et al. [24] into the repair framework DoubleR and make the following contributions from an applied perspective:

—We provide numerical analysis to show that repair layering significantly reduces the cross-rack repair traffic compared to state-of-the-art regenerating codes. We also provide reliability analysis to study the trade-off between the minimized cross-rack repair traffic and the reduced rack-level fault tolerance. Although similar numerical and reliability analysis on erasure-coded storage has been found in the literature (e.g., Huang et al. [25], Li et al. [31], and Sathiamoorthy et al. [31]), our analysis is new by specifically addressing the hierarchical nature of data centers.
—We propose two families of constructions of DRC. Both constructions preserve the theoretical guarantees of minimizing the cross-rack repair traffic and are specifically designed for the practical deployment in real-world data centers.
—We implement a DoubleR prototype atop Facebook's Hadoop Distributed File System (HDFS) [1]. We extensively parallelize the operations to mitigate repair overhead. We also export APIs that can incorporate not only DRC but also existing regenerating codes.
—We conduct testbed experiments on evaluating different erasure codes using our DoubleR prototype. We show that DRC increases the single failed node recovery throughput and reduces the degraded read time to an unavailable block. Our results also conform to the numerical results of bandwidth savings of DRC.

The remainder of the article proceeds as follows. In Section 2, we introduce and motivate the design of DoubleR. In Section 3, we present analytical results on DRC. In Section 4, we propose two practical constructions of DRC. In Section 5, we describe the implementation details of DoubleR. In Section 6, we present experimental results. In Section 7, we review related work. In Section 8, we provide a detailed discussion on the design trade-offs of DRC, and finally in Section 9, we conclude the article.

2 DOUBLER OVERVIEW

2.1 Motivation

Practical data centers are susceptible to both *independent* and *correlated* node failures [12, 16]: independent node failures mean that each node fails independently due to individual events (e.g., disk/node crashes), whereas correlated node failures mean that multiple nodes fail simultaneously due to a common disastrous event (e.g., power outages or common switch failures). In practice, racks are treated as the major failure domains in which correlated node failures are likely to occur. To deploy erasure coding in data centers, existing approaches mostly adopt flat block placement by placing each block of a stripe in a distinct rack [16, 25, 33, 38, 40, 44]. This tolerates the same numbers of node failures and rack failures, and provides the maximum fault tolerance against both independent and correlated node failures.

Our rationale is that rack failures are much less common than node failures in practice [16, 33], so it is viable to tolerate fewer rack failures than node failures. Thus, instead of adopting flat block placement, we opt for hierarchical block placement and place multiple blocks in the same rack to minimize the cross-rack repair traffic at the expense of reduced rack-level fault tolerance (note that each block is still stored in a distinct node for the same node-level fault tolerance). Given the constrained bandwidth resources for cross-rack links (see Section 1), minimizing the cross-rack repair traffic allows fast repair, thereby reducing the downtime of unavailable blocks (i.e., improved availability) and the window of vulnerability (i.e., improved durability).

In fact, the use of hierarchical block placement is also found in existing production storage systems to mitigate the cross-rack transfer overhead. For example, HDFS [49], which is replication based, by default places two replicas in one rack and one replica in a different rack. Quantcast File System (QFS) [34], which supports erasure coding, provides an option called *in-rack placement* to

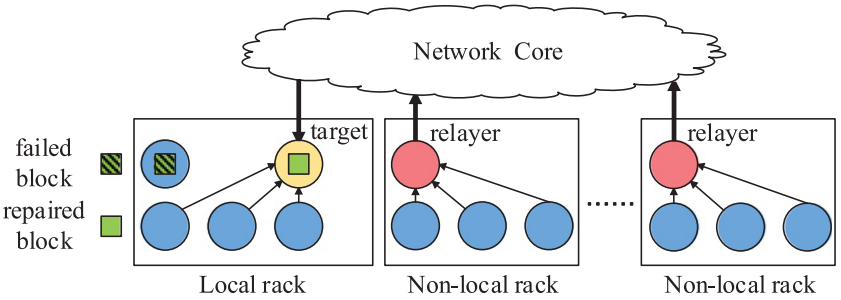


Fig. 1. Repair in DoubleR.

place multiple blocks in the same rack. An open issue is how to exploit the property of hierarchical block placement to minimize the cross-rack repair traffic in erasure-coded storage, and this is the key motivation of this work.

2.2 DoubleR Architecture

Given that multiple blocks are available in a rack, DoubleR exploits a two-layer repair approach by first repairing as much failed data as possible within each rack and then combining the partially repaired results from multiple racks to repair all failed data. To achieve bandwidth savings, DoubleR follows regenerating codes [15] by partitioning a block into smaller *subblocks* and allowing each node to compute encoded subblocks from its stored block during a repair operation. DoubleR takes one step further by *re-encoding* the encoded subblocks from all nodes in the same rack to achieve additional bandwidth savings across racks beyond regenerating codes.

Figure 1 illustrates the repair workflow in DoubleR. Specifically, we consider a hierarchical data center that is composed of multiple racks, each of which contains multiple storage nodes. Multiple nodes within the same rack are connected by a top-of-rack switch, whereas multiple racks are connected by an abstraction of switches called *network core* [11]. Repairing a failed block is done by retrieving available blocks from other nodes that reside in the same rack (called *local rack*) and different racks (called *nonlocal racks*). DoubleR selects one *target* node in the local rack to be responsible for reconstructing the failed block. It also selects one *relayer* node in each nonlocal rack to aggregate and forward the repaired results in that rack; typically, a relayer should be one of the nodes that locally stores an available block for the repair to save inner-rack network transfer. In each nonlocal rack, each node sends encoded subblocks to the relayer, which re-encodes the encoded subblocks. Multiple relayers then send the re-encoded subblocks across racks to the target, which reconstructs the failed block. In the deployment of DoubleR, we assign different relayers and targets for repairing multiple failed blocks (e.g., when recovering all lost data of a failed node) to harness parallelism (see Section 5).

DoubleR builds on the notion called *repair layering*, which decomposes a repair operation into different layers (nodes and racks in our case) along the hierarchy of a data center. By doing so, we can effectively mitigate the critical resource overhead (i.e., cross-rack repair traffic).

3 DOUBLE REGENERATING CODES

DoubleR builds on DRC [24] to realize repair layering. In this section, we define the notation and terminologies of erasure coding in the context of data centers and summarize the theoretical findings of DRC. We identify the connections between DRC and regenerating codes for special cases. We further compare DRC with existing erasure codes for more general cases through the

numerical analysis of bandwidth savings and the reliability modeling of mean time to data loss (MTTDL).

3.1 Background

We elaborate the background of erasure coding based on our discussion in Section 1. As multiple stripes are independently erasure coded, our discussion focuses on a single stripe. Specifically, we construct an erasure code, denoted by an (n, k, r) code, with three configurable parameters n, k (where $k < n$), and r (where $r \leq n$). For each stripe, we encode k original uncoded blocks of size B each into n coded blocks of the same size. For node-level fault tolerance, we distribute the coded blocks (or blocks in short) across n nodes (i.e., one block per node) that evenly reside in r racks with n/r nodes each. Here, we assume that n/r is an integer. For flat block placement, which is used by most erasure coding deployments, we have $r = n$, whereas our work addresses $r \leq n$. Unlike previous studies that typically construct an erasure code by two parameters n and k only, our work introduces the parameter r to take into account rack-level fault tolerance.

We focus on erasure codes that are *maximum distance separable* (MDS), meaning that any k out of n blocks suffice to reconstruct original uncoded data. MDS codes are *storage optimal*, meaning that they minimize storage redundancy (i.e., n/k times the original data size). Examples of MDS codes include Reed-Solomon (RS) codes [42], which have been widely deployed in production storage systems [2, 16, 33, 34], as well as minimum storage regenerating (MSR) codes [15], which minimize the repair traffic subject to the minimum storage redundancy. In this article, when we perform comparisons with regenerating codes on repair performance, we focus on MSR codes. Note that some non-MDS codes are also proposed to mitigate the repair traffic at the expense of higher storage redundancy. Examples include minimum bandwidth regenerating (MBR) codes [15] and locally repairable codes [25, 44].

In addition, we focus on *systematic* codes, meaning that k out of n coded blocks are in original uncoded form. As opposed to nonsystematic codes (i.e., all blocks are in coded form), systematic codes allow a storage system to directly access data without decoding. We refer to the k uncoded blocks as *data blocks*, whereas the remaining $n - k$ coded blocks are referred to as *parity blocks*. For brevity, if the context is clear, we simply refer to both data and parity blocks as *blocks*.

Repair. As in previous studies [15, 25, 28, 38, 40], this article focuses on optimizing the single-failure repair, which refers to either repairing a single unavailable block of a stripe in a degraded read operation or repairing all blocks of multiple stripes in a single node (i.e., one block per stripe) in a node recovery operation. Single-failure repair is the most common repair scenario in practice [25, 39]. Suppose that the target repairs a single failed block. In classical RS codes [42], the target retrieves k blocks from k available nodes. Thus, the repair traffic of RS codes per failed block (of size B) is

$$B \cdot k. \tag{1}$$

MSR codes [15] minimize the repair traffic while achieving the same minimum storage redundancy as RS codes (i.e., MDS). To repair a single failed block, each of the $n - 1$ available nodes can partition a block into $n - k$ subblocks and send an encoded subblock of size $B/(n - k)$ to the target.[1] The repair traffic of MSR codes per failed block [15] (which is provably minimum) is

$$B \cdot \frac{n-1}{n-k}. \tag{2}$$

[1] MSR codes allows fewer than $n - 1$ available nodes to send encoded information for repair, at the expense of higher repair traffic.

For data centers, our objective is to minimize the cross-rack repair traffic, which is the major bottleneck in a data center (see Section 1). Since each rack stores n/r blocks, this work mainly addresses the case where (1) $n/r \leq k$ and (2) $n/r \leq n-k$. Case (1) states that each rack has at most k blocks, implying that repairing a failed block must retrieve at least one available block across racks. Case (2) states that each rack has at most $n-k$ blocks, implying that a single rack failure does not introduce data loss; in other words, a data center can tolerate at least a single rack failure. DRC [24] is shown to achieve the minimum cross-rack repair traffic per failed block, given by:

$$B \cdot \frac{r-1}{r-\lfloor kr/n \rfloor}. \tag{3}$$

If we distribute blocks across $r = n$ racks as in flat block placement, Equation (3) reduces to the minimum repair traffic of MSR codes in Equation (2).

Connections with regenerating codes. We point out that the minimum cross-rack repair traffic in Equation (3) can be achieved by MSR codes for specific settings of parameters, as shown in the following theorem.

Theorem 3.1. *MSR codes can achieve the minimum cross-rack repair traffic for general n and k with $r = \frac{n}{n-k}$, assuming that n is divisible by $n-k$ (i.e., r is an integer).*

Proof. We deploy MSR codes following hierarchical block placement by setting $r = \frac{n}{n-k}$, so each rack has $n-k$ blocks in $n-k$ different nodes. To repair a failed block, the target retrieves $n-k-1$ encoded subblocks from its local rack and $(r-1)(n-k)$ encoded subblocks from nonlocal racks. Since each encoded subblock has size $\frac{B}{n-k}$, the cross-rack repair traffic of MSR codes is $(r-1)(n-k) \cdot \frac{B}{n-k} = (r-1)B$.

Given that $r = \frac{n}{n-k}$, we can show that $k = n - \frac{n}{r}$ and hence $\lfloor \frac{kr}{n} \rfloor = \lfloor \frac{(n-n/r)r}{n} \rfloor = r-1$. Thus, we can express the minimum cross-rack repair traffic in Equation (3) as $B \cdot \frac{r-1}{r-(r-1)} = (r-1)B$. In other words, the cross-rack repair traffic of MSR codes is in fact the minimum. □

3.2 Examples

Based on Equation (3), we provide examples to motivate how DRC reduces the cross-rack repair traffic over MSR codes. Here, we fix $n = 6$ and $k = 3$ in our examples.

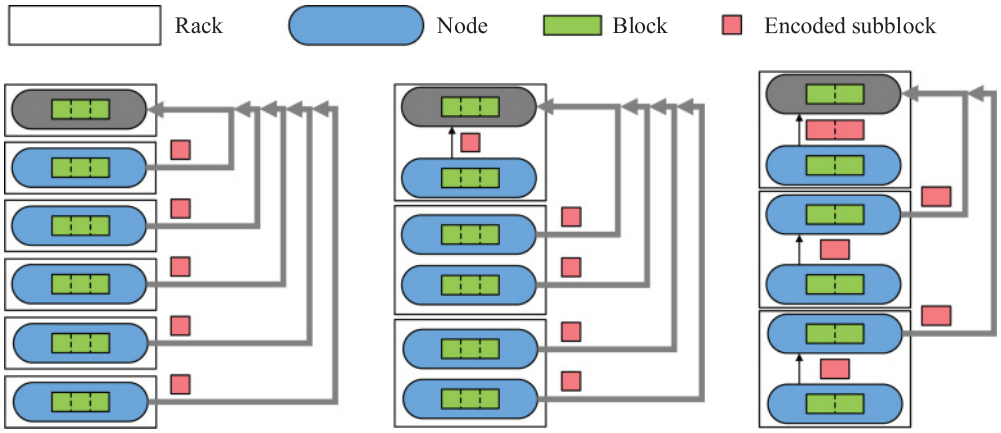
Suppose that we deploy MSR codes using flat block placement. We set $(n, k, r) = (6, 3, 6)$ and denote the code by MSR(6,3,6). Figure 2(a) shows the block placement of MSR(6,3,6). To repair a failed block, each node in a distinct rack partitions its stored block into $n-k = 3$ subblocks and sends one encoded subblock of size $\frac{B}{n-k} = B/3$ [15]. From Equation (3), the cross-rack repair traffic of MSR(6,3,6) is $5B/3$.

Clearly, we can also deploy regenerating codes using hierarchical block placement. We set $(n, k, r) = (6, 3, 3)$ by placing two blocks per rack and denote the code by MSR(6,3,3). Figure 2(b) shows how MSR(6,3,3) repairs a failed block. We see that the target can retrieve one encoded subblock from the local rack, whereas it still needs to retrieve four encoded subblocks from nonlocal racks. Thus, the cross-rack repair traffic of RC(6,3,3) is reduced to $4B/3$.

DRC takes advantage of hierarchical block placement by re-encoding the encoded subblocks in the relayer of each rack. We again set $(n, k, r) = (6, 3, 3)$ and denote the code by DRC(6,3,3). Figure 2(c) shows how DRC(6,3,3) repairs a failed block. From Equation (3), the cross-rack repair traffic of DRC(6,3,3) is further reduced to B.

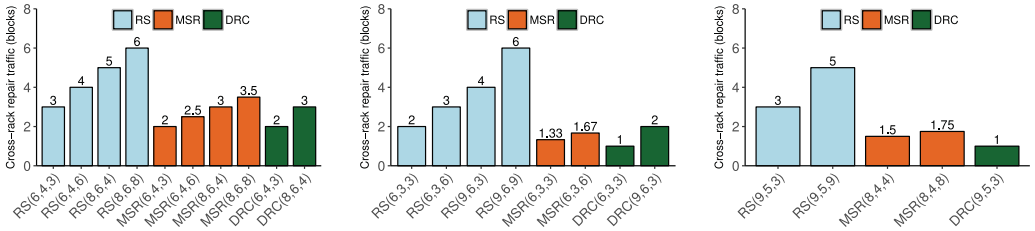
3.3 Numerical Analysis

We present numerical results to demonstrate the benefits of DRC in minimizing the cross-rack repair traffic for different cases of (n, k, r). We consider the following erasure codes:



(a) MSR(6,3,6) (b) MSR(6,3,3) (c) DRC(6,3,3)

Fig. 2. Motivating examples.



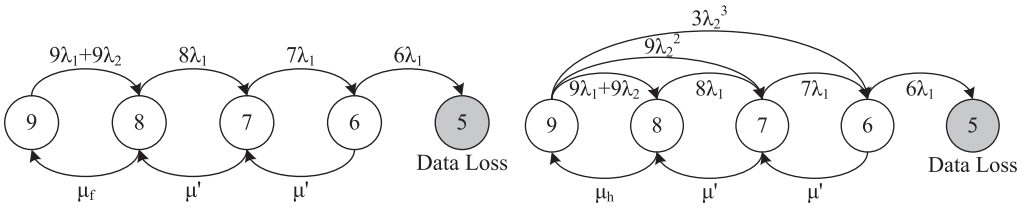
(a) $n-k=2$ (b) $n-k=3$ (c) $n-k=4$

Fig. 3. Numerical results of cross-rack repair traffic (in blocks) for repairing a failed block under different erasure code configurations, grouped by the same $n-k$.

—*RS*: To repair a failed block in RS codes [42], the target retrieves k available blocks. If $r < n$ (i.e., hierarchical block placement), we assume that the target first retrieves $n/r - 1$ available blocks from the local rack, followed by retrieving the remaining $k - (n/r - 1)$ blocks from nonlocal racks, to make the cross-rack repair traffic as low as possible for RS codes. We use RS codes as the baseline.
—*MSR*: We consider two parameter settings whose systematic MSR code constructions have been proposed in the literature: (1) $n - k = 2$ [35, 52] and (2) $n = 2k$ [38, 41, 47]. Note that for $n - k = 2$ and $r = n/2$, MSR codes achieve the same cross-rack repair traffic as DRC (see Theorem 3.1). We include them for completeness.
—*DRC*: We consider two parameter settings: (1) general (n, k) with $r = n/(n - k)$ and (2) $(n, k, r) = (3z, 2z - 1, 3)$ for $z \geq 2$. In Section 4, we provide code constructions for both settings.

Figure 3 shows the numerical results of cross-rack repair traffic (in units of blocks) for repairing a failed block under different configurations of erasure codes, which we group by the same $n - k$ (i.e., the same number of node failures that can be tolerated). We make the following observations:

—As expected, there is a storage-bandwidth trade-off. For a given code with the same $n - k$, the cross-rack repair traffic increases when the storage redundancy (i.e., n/k) decreases. For



(a) Flat block placement $(n, k, r) = (9, 6, 9)$ (b) Hierarchical block placement: $(n, k, r) = (9, 6, 3)$

Fig. 4. Markov models for flat block placement and hierarchical block placement.

example, RS(8,6,8) has 11.3% less storage redundancy than RS(6,4,6) but 50% higher cross-rack repair traffic.

- For a given code and the same n and k, hierarchical block placement reduces the cross-rack repair traffic over flat block placement at the expense of reduced rack-level fault tolerance. For example, RS(6,4,3) incurs 25% less cross-rack repair traffic than RS(6,4,6); MSR(6,4,3) incurs 20% less cross-rack repair traffic than MSR(6,4,6).
- For the same (n, k, r), DRC incurs less cross-rack repair traffic than RS codes. The percentage reduction increases with $n - k$. For example, DRC(9,5,3) incurs 66.7% less cross-rack repair traffic than RS(9,5,3).
- For $n - k \geq 3$, DRC incurs not only less cross-rack repair traffic but also less storage redundancy than MSR codes. For example, DRC(9,5,3) incurs 33.3% less cross-rack repair traffic and 20% less storage redundancy than MSR(8,4,4).

3.4 Reliability Analysis

Recall that DRC leverages hierarchical block placement to trade rack-level fault tolerance for the minimum cross-rack repair traffic. We now study the reliability trade-off of DRC due to hierarchical block placement. Here, we analyze the MTTDL metric via the standard Markov modeling, as used by many previous studies (e.g., Cidon [12], Ford et al. [16], Huang et al. [25], Sathiamoorthy et al. [44], and Silberstein [50]). Although the effectiveness of Markov-based reliability analysis is debatable [21], we believe that it suffices for providing preliminary insights on reliability for this work.

Model. In our analysis, we fix $n = 9$ and $k = 6$ (which are also used by QFS [34]). Figure 4 shows the Markov models for flat block placement with $(n, k, r) = (9, 6, 9)$ and hierarchical block placement with $(n, k, r) = (9, 6, 3)$ (in Section 4.2, we provide a DRC construction for $(9, 6, 3)$). Suppose that we distribute blocks of multiple stripes across n nodes. Each state represents the number of available nodes—for instance, *State 9* means that all nodes are healthy, whereas *State 5* implies data loss. We assume that interfailure and interrepair times are exponentially distributed.

We model both independent and correlated node failures (see Section 2.1). For independent node failures, let λ_1 be the independent failure rate of each node. The state transition rate from State i to State $i - 1$, where $6 \leq i \leq 9$, is $i\lambda_1$, since any of the i nodes in State i fail independently. For correlated node failures, we consider the scenario where each rack (the largest failure domain in our case) experiences a power outage that brings down a fixed fraction of nodes simultaneously [12, 49]. In our modeling, we assume that each node fails with a probability that is equal to the fraction of nodes being brought down by a power outage. Let λ_2 be the failure rate of each node due to correlated node failures. We assume that correlated node failures are rare and only occur when all nodes are healthy (i.e., State 9); in other words, a data center operating in degraded mode is only subject to independent node failures. This assumption also simplifies our analysis. Thus,

Table 1. MTTDLs of Flat Block Placement and Hierarchical Block Placement for Different Values of $1/\lambda_1$ (Years) and $\gamma_1 = 1$ Gb/s

$1/\lambda_1$ (years)	2	4	6	8	10
Flat, without correlated	2.56E+06	4.08E+07	2.06E+08	6.52E+08	1.59E+09
Flat, with correlated	2.54E+06	4.00E+07	2.00E+08	6.27E+08	1.51E+09
Hierarchical, without correlated	3.41E+06	5.44E+07	2.75E+08	8.69E+08	2.12E+09
Hierarchical, with correlated	3.28E+06	4.69E+07	1.96E+08	4.81E+08	8.80E+08

for flat block placement (see Figure 4(a)), the state transition rate from State 9 to State 8 adds $9\lambda_2$, as each node residing in a distinct rack can fail due to correlated node failures. For hierarchical block placement (see Figure 4(b)), there are three cases: (1) from State 9 to State 8, the state transition rate adds $3 \cdot (3\lambda_2) = 9\lambda_2$, as a node failure can occur in any one of three nodes in any one of the three racks; (2) from State 9 to State 7, the state transition rate is $3 \cdot (3\lambda_2^2) = 9\lambda_2^2$, since a two-node failure can occur in any two of three nodes in any one of the three racks; and (3) from State 9 to State 6, the state transition rate is $3\lambda_2^3$ since a three-node failure can occur in any one of the three racks.

To model repair, we assume that the repair times are proportional to the amount of repair traffic. When there is only one single failed node, let μ_f and μ_h be the repair rates of a failed node from State 8 to State 9 in flat block placement and hierarchical block placement, respectively. When there are multiple failed nodes, we assume that we repair one node at a time (similar to the analysis in Huang et al. [25] and Sathiamoorthy et al. [44]) and the repair is done by retrieving the size of original data in both placement schemes, and let μ' be the repair rate for each node from State i to State $i+1$, where $6 \leq i \leq 8$.

We can configure the parameters as follows. For λ_1, we assume that the mean time to failure (MTTF) of a node is in the range of a few years [45] (e.g., $1/\lambda_1 = 4$ years [44]). For λ_2, we follow the assumption that a power outage occurs once a year and 0.5% to 1% of nodes fail after a power outage [49]; in this case, the MTTF of a node due to a power outage is $0.5\% \leq \lambda_2 \leq 1\%$ (per year). For repair, let γ be the available cross-rack bandwidth, S be the storage capacity of a node, and C be the repair traffic per unit of repaired data. For example, for a single-node repair in flat block placement, $C = 8/3$ if MSR codes are used (see Equation (2)), so $\mu_f = \gamma/(8S/3)$. For a single-node repair in hierarchical block placement, $C = 2$ if DRC is used (see Equation (3)), so $\mu_h = \gamma/(2S)$. When there are multiple failed nodes, each failed node is repaired from the available blocks of any k nodes (i.e., the MDS property), so $C = k = 6$ and $\mu' = \gamma/(6S)$.

Analysis. We now evaluate the MTTDLs of both block placement schemes. We consider the scenarios with (1) independent node failures only (i.e., $\lambda_2 = 0$) and (2) both independent and correlated node failures, in which we set $\lambda_2 = 0.5\%$ (per year). We also fix $S = 1$TiB.

We show the MTTDL results for two parameter settings: (1) we fix $\gamma = 1$Gb/s [44] and vary $1/\lambda_1$ from 2 to 10 years (see Table 1), and (2) we fix $1/\lambda_1 = 4$ years [44] and vary γ from 200Mb/s to 2Gb/s (see Table 2). Overall, with independent node failures only, hierarchical block placement achieves higher MTTDL than flat block placement (by around 33%) due to the minimized cross-rack repair traffic in repairing a single node failure. However, when correlated node failures are included, the MTTDL drop in hierarchical block placement is much more obvious than in flat block placement.

Specifically, in the presence of correlated node failures, hierarchical block placement has a relatively higher MTTDL than flat block placement when independent node failures are more frequent, in which case the repair rate plays a more dominant role in MTTDL. For example, in Table 1, when $1/\lambda_1 = 2$ years and there are correlated node failures, the MTTDL of flat block

Table 2. MTTDLs of Flat Block Placement and Hierarchical Block Placement for Different Values of γ (Gb/s) and $1/\lambda_1 = 4$ Years

γ (Gb/s)	0.2	0.5	1	2
Flat, without correlated	3.32E+05	5.12E+06	4.08E+07	3.26E+08
Flat, with correlated	3.26E+05	5.02E+06	4.00E+07	3.19E+08
Hierarchical, without correlated	4.42E+05	6.82E+06	5.44E+07	4.34E+08
Hierarchical, with correlated	4.25E+05	6.33E+06	4.69E+07	3.09E+08

placement is 2.54×10^6 years, whereas that of hierarchical block placement is 3.28×10^6 years (29% higher). However, hierarchical block placement has less MTTDL than flat block placement when $1/\lambda_1$ increases or γ increases (e.g., $1/\lambda_1 \geq 6$ years in Table 1 and $\gamma = 2$Gb/s in Table 2). In this case, the improvement due to the minimized cross-rack repair traffic becomes less important. Nevertheless, the overall impact of failures is also low and hierarchical block placement already achieves a fairly high MTTDL (e.g., over 10^8 years for $1/\lambda_1 \geq 6$ years as shown in Table 1).

4 PRACTICAL DRC CONSTRUCTIONS

It is shown in Hu et al. [24] that DRC can be constructed via random linear codes. A major drawback is that such a construction is not practical, as it is nonsystematic (i.e., all blocks are in coded form). This implies that extra decoding is needed to access any coded block. In this section, we provide practical DRC constructions that are suitable for real deployment.

4.1 Goals

Our practical DRC constructions aim for several design goals:

(1) *Theoretical guarantees*: Each construction is MDS (and hence storage optimal) and minimizes the cross-rack repair traffic.
(2) *Systematic*: The original data blocks are kept after encoding.
(3) *Exact-repair*: Each reconstructed block has the same content as the original failed block.
(4) *Small finite fields*: The arithmetic operations of encoding are done over the Galois field GF(2^8); in other words, the encoding can be done in units of bytes [20].
(5) *Small redundancy*: Each construction can achieve storage redundancy below 2×.
(6) *Polynomial number of subblocks per block*: The number of subblocks per block is polynomial with respect to the number of original data blocks of a stripe (i.e., k); this reduces the access overhead to subblocks.
(7) *Reduced inner-rack repair traffic*: The amount of traffic that a relayer receives from all available nodes within the same rack is no more than that it sends out to the target across the racks.
(8) *Balanced cross-rack repair traffic*: Each relayer sends the same amount of cross-rack traffic during repair.

The design goals have the following implications. Goal 1 ensures that our practical DRC constructions preserve the theoretical guarantees as proven in Hu et al. [24]. Goals 2 through 4 improve nonsystematic regenerating codes [15] and DRC [24], both of which require that the Galois field size needs to be sufficiently large to provide theoretical guarantees. Goal 5 improves existing systematic regenerating codes including MISER codes [47], Product-Matrix (PM) codes [41], and PM-RBT codes [38], all of which require the redundancy be at least 2×. Goal 6 improves Butterfly codes [35], which have 2^{k-1} subblocks per block (i.e., exponential with k). Goal 7 ensures that by

limiting the inner-rack repair traffic, the cross-rack repair traffic is the most dominant factor in the repair performance. Finally, Goal 8 ensures that the repair operation is load balanced across all racks.

To this end, we propose two families of practical DRC constructions for different possible configurations of (n, k, r). Both families can tolerate a variable number of node failures, although tolerating only a single-rack failure to trade for the minimum cross-rack repair traffic. Family 1, referred to as DRC$(n, k, \frac{n}{n-k})$, supports general n and k ($k < n$) with $r = \frac{n}{n-k}$, provided that r is an integer. It can be configured with low storage redundancy (e.g., 1.33× for (8, 6, 4)). Family 2, referred to as DRC$(3z, 2z-1, 3)$, supports any integer $z \geq 2$. Its redundancy is 1.5 to 2×, which is generally higher than that of Family 1 but achieves less cross-rack repair traffic. Our current DoubleR prototype has implemented DRC(6,4,3), DRC(8,6,4), and DRC(9,6,3) for Family 1, and DRC(6,3,3) and DRC(9,5,3) for Family 2.

One key property of both families is that their encoding/decoding operations are based on the classical RS codes [42], which have been well studied and widely deployed in production [2, 16, 33, 34]. Thus, we can exploit the theoretical guarantees provided by RS codes (e.g., the MDS property). A key challenge is how to augment RS codes to satisfy the design goals listed in Section 4.1, which we address in the following.

Before we present the two families of DRC constructions, we remark that several recent studies [19, 43, 57, 58] have proposed MSR code constructions that support general n and k and also achieve Goals 1 through 6 stated earlier. Recall that Family 1 has the same parameters in Theorem 3.1. Thus, we can directly use the recently proposed MSR code constructions to minimize the cross-rack repair traffic. Nevertheless, such MSR code constructions are mainly studied from a theoretical standpoint, and their implementations and evaluations are still open issues. In contrast, Family 1 can be realized by RS codes, which are well known in practice.

4.2 Family 1: DRC$(n, k, \frac{n}{n-k})$

Family 1 integrates RS codes with *interference alignment* [47], which makes the reduction of inner-rack repair traffic (Goal 7) and the balance of cross-rack repair traffic across multiple racks (Goal 8) possible. We use DRC(9,6,3) as an example, as shown in Figure 5(a).

Construction. For each stripe, we collect a set of k data blocks and divide each block into $n-k$ subblocks, called *data subblocks*. We group the subblocks at the same offset of all data blocks into a set, so there are $n-k$ sets in total. We construct $n-k$ parity blocks from the k data blocks by encoding each set of k data subblocks into $n-k$ coded subblocks, called *parity subblocks*, using RS encoding.

For example, consider DRC(9,6,3) in Figure 5(a). We have nine nodes, denoted by N_1, N_2 , . . . , N_9, that are placed across three racks, denoted by R_1, R_2, and R_3. We divide $k = 6$ data blocks into three sets of subblocks $\{a_1, \ldots, a_6\}$, $\{b_1, \ldots, b_6\}$, and $\{c_1, \ldots, c_6\}$, which are stored in the first six nodes, N_1 to N_6. We perform RS encoding to encode each of the three sets of data subblocks and respectively form three sets of parity subblocks, denoted by $\{p_1, p_2, p_3\}$, $\{p_4, p_5, p_6\}$, and $\{p_7, p_8, p_9\}$. The parity blocks are stored in the remaining three nodes, N_7 to N_9. Since the original k data blocks can be reconstructed from any k blocks of a stripe via RS decoding, the MDS property is preserved.

Repair idea. We first describe the main idea of repairing a data block and discuss how it also applies to repairing a parity block.

Without loss of generality, we repair a data block in N_1. Each relayer (say, N_4 and N_7) sends the target, denoted by $\underline{N}_1$, three encoded subblocks of size $B/3$ each. Based on DoubleR, the target $\underline{N}_1$ can obtain data from N_2 and N_3 of the local rack and relayers N_4 and N_7 of the nonlocal racks to repair the lost subblocks $\{a_1, b_1, c_1\}$. Obviously, N_2, N_3, and N_4 cannot provide any information

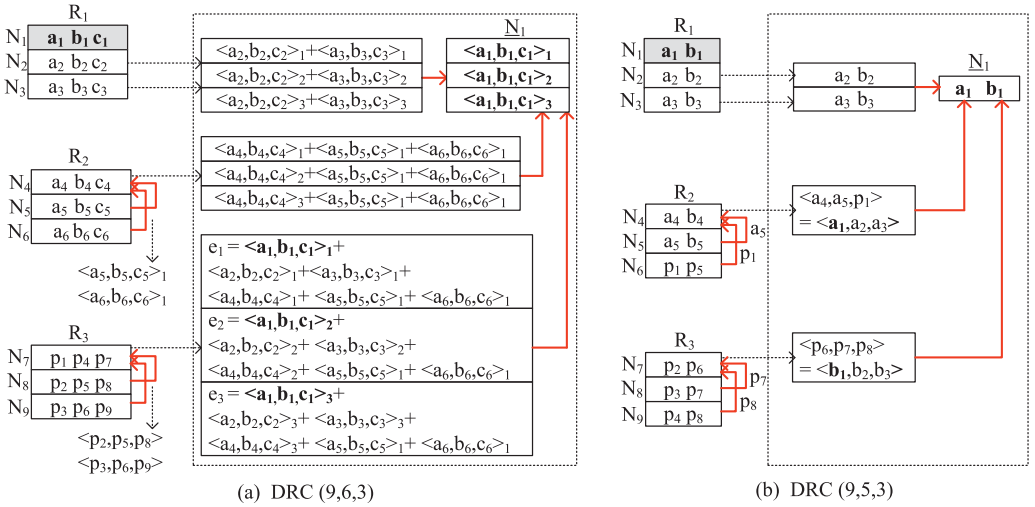


Fig. 5. Two families of DRC. The red (solid) arrows refer to the network transfers of encoded subblocks.

pertaining to $\{a_1, b_1, c_1\}$, whereas only N_7 can provide three encoded subblocks, denoted by $\{e_1, e_2, e_3\}$, in the form of linear combinations of all data subblocks. Thus, $\underline{N}_1$ must be able to solve for $\{a_1, b_1, c_1\}$ from $\{e_1, e_2, e_3\}$ by subtracting out the "interference" information pertaining to the nonfailed data subblocks. We borrow the idea of interference alignment [47] to make the interference information formed by a number of *aligned* linear combinations, which are either the same or scalar multiples of each other. Instead of solving for individual subblocks, the repair operation now cancels out aligned linear combinations through linear algebra.

To repair a parity block (e.g., in N_7), we represent its parity subblocks p_1, p_4, p_7 by a'_1, b'_1, and c'_1, respectively. Due to RS encoding, the data subblocks a_1, b_1, and c_1 are in fact the linear combinations of $\{a'_1, a_2, \ldots, a_6\}$, $\{b'_1, b_2, \ldots, b_6\}$, and $\{c'_1, c_2, \ldots, c_6\}$, respectively. Thus, we can view $\{a'_1, b'_1, c'_1\}$ as data subblocks and $\{a_1, b_1, c_1\}$ as parity subblocks. In this way, we can apply the same approach of repairing a data block into repairing a parity block.

Repair details. We specify the detailed steps of repairing N_1, whereas the same methodology applies to other nodes:

(1) The relayer N_7 sends encoded subblocks $\{e_1, e_2, e_3\}$ to the target $\underline{N}_1$ such that each encoded subblock comprises the same aligned linear combination of $\{a_5, b_5, c_5, a_6, b_6, c_6\}$. We elaborate how to form $\{e_1, e_2, e_3\}$ in the following.

(1.1) e_1 is simply given by $p_1 + p_4 + p_7$, which can be viewed as a linear combination of all data subblocks. Let $e_1 = \sum_{i=1}^{6} \langle a_i, b_i, c_i \rangle_1$, where $\langle x_1, \ldots, x_m \rangle_j$ denotes the j^{th} linear combination of subblocks $\{x_1, \ldots, x_m\}$.

(1.2) e_2 is a linear combination of $\{p_1, p_4, p_7, \langle p_2, p_5, p_8 \rangle\}$, where $\langle p_2, p_5, p_8 \rangle$ is a linear combination sent from N_8 to N_7. We ensure that e_2 also contains $\sum_{i=5}^{6} \langle a_i, b_i, c_i \rangle_1$, which aligns with part of e_1, and is represented as $e_2 = \sum_{i=1}^{4} \langle a_i, b_i, c_i \rangle_2 + \sum_{i=5}^{6} \langle a_i, b_i, c_i \rangle_1$. This can be accomplished by tuning the coding coefficients as follows. Let $e_2 = \gamma_1 p_1 + \gamma_2 p_4 + \gamma_3 p_7 + (\gamma_4 p_2 + \gamma_5 p_5 + \gamma_6 p_8)$, where γ_i's are some coding coefficients. Then we can tune γ_1 and γ_4 in such a way that $\gamma_1 p_1 + \gamma_4 p_2$ have the same terms for a_5 and a_6 as in $\sum_{i=5}^{6} \langle a_i, b_i, c_i \rangle_1$. Similarly, we can tune γ_2 and γ_5 to have the same terms for b_5 and b_6, and tune γ_3 and γ_6 to have the same terms for c_5 and c_7.

(1.3) e_3 is a linear combination of $\{p_1, p_4, p_7, \langle p_3, p_6, p_9 \rangle\}$, where $\langle p_3, p_6, p_9 \rangle$ is a linear combination sent from N_9 to N_7. We also ensure that e_3 also contains $\sum_{i=5}^{6} \langle a_i, b_i, c_i \rangle_1$, which again aligns with

part of e_1, by setting $e_3 = \sum_{i=1}^{4}\langle a_i, b_i, c_i\rangle_3 + \sum_{i=5}^{6}\langle a_i, b_i, c_i\rangle_1$. This can be done by tuning coding coefficients as earlier.

(2) The relayer N_4 computes $\langle a_4, b_4, c_4\rangle_1$ and retrieves the linear combinations $\langle a_5, b_5, c_5\rangle_1$ and $\langle a_6, b_6, c_6\rangle_1$ from N_5 and N_6, respectively. It sends $\underline{N}_1$ three encoded subblocks, each of which aligns with part of e_1, e_2, or e_3. For example, the linear combination $\sum_{i=4}^{6}\langle a_i, b_i, c_i\rangle_1$ aligns with part of e_1.

(3) Each of the nodes N_2 and N_3 in the local rack sends $\underline{N}_1$ three encoded subblocks, each of which aligns with part of e_1, e_2, or e_3.

(4) $\underline{N}_1$ cancels out the aligned linear combinations. It now has $\langle a_1, b_1, c_1\rangle_1$, $\langle a_1, b_1, c_1\rangle_2$, and $\langle a_1, b_1, c_1\rangle_3$, which can be used to solve for $\{a_1, b_1, c_1\}$.

4.3 Family 2: DRC$(3z, 2z-1, 3)$

Family 2 differs from Family 1 by allowing a node in a nonlocal rack to merely read a subblock from its local storage and send it to the relayer, without performing encoding operations. It follows the spirit of repair by transfer [46] and helps reduce disk I/Os. We use DRC(9,5,3) as an example, as shown in Figure 5(b).

Construction. For each stripe, we collect k data blocks and divide each block into two subblocks (i.e., $2(2z-1)$ data subblocks in total). We group the subblocks at the same offset of all data blocks into a set (i.e., there are two sets in total). Each set of $2z-1$ data subblocks is independently encoded using RS codes to generate $z+1$ parity subblocks. We distribute the $n = 3z$ blocks across three racks, each of which contains z blocks (i.e., $3z$ subblocks in total). Like Family 1, since each set of subblocks is encoded with RS codes, the MDS property is preserved.

For example, consider DRC(9,5,3) in Figure 5(b) (i.e., $z = 3$). We have nine nodes $N_1, N_2, \ldots, N_9$ that are placed across three racks R_1, R_2, and R_3. The data blocks have two sets of data subblocks $\{a_1, \ldots, a_5\}$ and $\{b_1, \ldots, b_5\}$, and we place the data blocks in N_1 to N_5. We encode them using RS codes to generate two sets of parity subblocks $\{p_1, p_2, p_3, p_4\}$ and $\{p_5, p_6, p_7, p_8\}$, respectively, and place the parity blocks in N_6 to N_9.

Repair idea. As in Family 1, we only need to consider how to repair a data block (e.g., in N_1), whereas we apply the same methodology to repair a parity block. Our observation is that each failed subblock can be reconstructed by $2z-1$ subblocks from two racks only—that is, the $z-1$ subblocks of the same set in the local rack and the z subblocks of the same set in one of the nonlocal racks. For example, in Figure 5(b), the failed subblock a_1 can be reconstructed from $\{a_2, a_3, a_4, a_5, p_1\}$, which reside in R_1 and R_2, while the failed subblock b_1 can be reconstructed from $\{b_2, b_3, p_6, p_7, p_8\}$, which reside in R_1 and R_3. The two relayers (say, N_4 and N_7) only need to send information that is needed for reconstructing a_1 and b_1, respectively, and the cross-rack repair traffic can be shown to be minimum. In addition, note that each of $N_4, \ldots N_9$ only needs to read a subblock from its local storage, where the subblock size is only half of the block size. This reduces disk I/Os.

Repair details. We specify the detailed steps of repairing the data block in N_1:

(1) The relayer N_4 in R_2 collects the subblocks a_4, a_5, and p_1 within the same rack. It computes a linear combination of the collected subblocks as an encoded subblock such that both a_4 and a_5 can be canceled out; in other words, it computes the encoded subblock as $\langle a_4, a_5, p_1\rangle = \langle a_1, a_2, a_3\rangle$. This can be accomplished by simply subtracting out a_4 and a_5 from p_1, as p_1 is a linear combination of $\{a_1, a_2, \ldots, a_5\}$. It sends the encoded subblock to the target $\underline{N}_1$.

(2) The relayer N_7 in R_3 collects the subblocks p_6, p_7, and p_8 within the same rack. Similar to earlier, N_7 computes a linear combination of p_6, p_7, and p_8 as an encoded subblock such that both

b_4 and b_5 can be canceled out. Thus, the encoded subblock is computed as $\langle p_6, p_7, p_8 \rangle = \langle b_1, b_2, b_3 \rangle$ and sent to the target $\underline{N}_1$.

(3) Both nodes N_2 and N_3 send their stored subblocks to the target $\underline{N}_1$.

(4) $\underline{N}_1$ solves for a_1 and b_1 by canceling out a_2, b_2, a_3, and b_3 from $\langle a_1, a_2, a_3 \rangle$ and $\langle b_1, b_2, b_3 \rangle$ through linear algebra.

5 IMPLEMENTATION

We implement DoubleR on Facebook's HDFS [1], which integrates HDFS-RAID [2] to support erasure coding atop HDFS [49]. We provide an overview of how HDFS realizes erasure coding and then describe how we implement DoubleR atop HDFS.

5.1 HDFS Overview

HDFS organizes data as fixed-size data blocks, each of which is a basic unit of read/write operations and has a large size (e.g., 64MiB) to mitigate random access overhead. It comprises a single *NameNode* for managing file operations and multiple *DataNodes* for storing data.

HDFS-RAID adds a *RaidNode* to HDFS for managing erasure-coded blocks. The RaidNode first stores data blocks with replication such that each data block has multiple copies stored in distinct nodes. It later transforms the blocks into erasure-coded blocks in the background. The RaidNode coordinates the transformation via MapReduce [14]. Specifically, to construct an erasure code with parameters n and k for a stripe, a map task of a MapReduce job collects k data blocks from different DataNodes, encodes them into $n - k$ parity blocks, and distributes the n blocks across n different DataNodes. In addition, the RaidNode periodically checks for any failed blocks and triggers the repair operation if needed.

5.2 DoubleR Details

We explain how we extend the Facebook's HDFS to include DoubleR and provide justifications for our design choices.

Erasure codes. We implemented different erasure codes based on the parameters shown in Section 3.3., including RS codes, MSR codes, and DRC. For MSR codes, we implemented Butterfly codes [35] for $n - k = 2$ and MISER codes [47] for $n = 2k$; both codes are systematic codes. For DRC, we implemented the two families of practical DRC constructions for different combinations of (n, k, r) (see Section 3).

Each erasure code is implemented in C++ using Intel's ISA-L [3]. We mainly use two ISA-L APIs: `ec_init_tables`, which specifies the coding coefficients, and `ec_encode_data`, which specifies the encoding/decoding operations. Both APIs automatically optimize the computations based on the hardware configurations (e.g., Intel SSE instructions are used if supported). We link each erasure code implementation with Hadoop via Java Native Interface (JNI).

Strip size. In the original HDFS-RAID, a block is partitioned into multiple *strips* for erasure coding such that the strips at the same block offsets are encoded together to form a smaller-size stripe. Our DoubleR implementation exploits this feature and further uses multithreading (see details in the following) to parallelize the encoding/decoding of blocks that now span multiple smaller-size stripes. For both regenerating codes and DRC, each strip is divided into *substrips* so that the available nodes can send encoded substrips for repair; in other words, an encoded subblock in regenerating codes and DRC is composed of multiple encoded substrips of a block. Note that if the strip size is too small, there will be heavy I/O access overhead. We study the impact of the strip size through experiments (see Section 6).

Block placement. DoubleR groups multiple blocks belonging to the same stripe in the same rack. We modify the RaidNode to specify how blocks of each stripe are stored based on the parameters (n, k, r).

Repair operations. DoubleR focuses on repairing single failures. It currently supports two types of repair operations: node recovery and degraded reads. A *node recovery* operation repairs multiple failed blocks of a single failed node, in which each failed block belongs to a different stripe. We modify the RaidNode to call DoubleR for node recovery when it detects a failed node. However, a *degraded read* operation repairs a single unavailable block. We modify the file system client to call DoubleR to perform a degraded read when it fails to access a block and triggers a block missing exception.

Parallelization. DoubleR does not leverage MapReduce for repair as in the original HDFS-RAID; instead, its implementation embodies extensive parallelization to speed up a repair operation and move the bottleneck to cross-rack transfer. First, we use multithreading at the node level to parallelize disk I/O, encoding/decoding, and network transfer operations. We also spawn multiple threads to repair multiple strips of a failed block in parallel. In addition, for node recovery, which involves the repair of multiple failed blocks of a single node, we assign different relayers and targets for different stripes to harness parallelism in a data center.

Exported APIs. DoubleR exports three primitive APIs for a repair operation: (1) `NodeEncode`, in which a storage node computes encoded subblocks from its locally stored block; (2) `RelayerEncode`, in which a relayer computes re-encoded subblocks from the encoded subblocks of the storage nodes in the same rack; and (3) `Decode`, in which a target reconstructs a failed block, using the blocks from the nodes in the same rack and blocks from the relayers in different racks. For regenerating codes [15] and their variants [35, 47], we only need to implement `NodeEncode` and `Decode`; for DRC, we implement all three APIs.

6 EXPERIMENTS

We present evaluation results on DoubleR from testbed experiments. We address the following questions. Can DRC achieve the theoretical performance (i.e., the numerical results in Section 3.3) in a real networked environment? Does minimizing cross-rack repair traffic play a key role in improving the overall repair performance?

6.1 Methodology

Testbed setup. Our testbed experiments are conducted on a cluster of 11 machines. Each machine has a quad-core 3.4-GHz Intel Core i5-3470, 16GiB RAM, and a Seagate ST1000DM003 7200 RPM 1-TiB SATA hard disk. All machines are interconnected via a 10-Gb/s Ethernet switch. We deploy Facebook's HDFS [1] on 10 machines. One machine runs both the NameNode and RaidNode, and each of the other n machines runs a DataNode for an (n, k, r) code, where n is up to 9 in our evaluation.

To mimic a hierarchical data center, we assign one dedicated machine called the *gateway* to mimic the network core in Figure 1. Specifically, we partition the n DataNodes into r logical racks. If one machine in a logical rack wants to send data to another machine in a different logical rack, its cross-rack traffic will first be redirected to the gateway, which then relays the traffic to the destination machine; otherwise, its inner-rack traffic will be sent directly to the destination machine through the 10Gb/s Ethernet. We configure the routing table of each machine using the Linux command `route` for the traffic redirection. In addition, we limit the outgoing bandwidth of the

gateway (i.e., the available cross-rack bandwidth) using the Linux traffic control command `tc` to mimic the oversubscription scenario (see Section 1).

Default parameters. We study different erasure codes that we implemented (see Section 5.2). By default, we configure the block size as 64MiB (which is also the default in Facebook's HDFS) and the strip size as 256KiB. One subtlety is that both MISER(6,3,3) and DRC(9,6,3) need to partition a block (strip) into three subblocks (substrips) for repair. To allow even partitioning, for both cases, we configure the block size as 63MiB and the strip size as 252KiB as their defaults. We also set the default gateway bandwidth as 1Gb/s to simulate the available cross-rack bandwidth for repair in production data centers [44]. We vary one of the parameters in each of our experiments. Our results are averaged over five runs; we omit the variances of the results, as they are insignificant based on our evaluation.

6.2 Microbenchmarks

Before we measure the node recovery and degraded read performance, we first show via microbenchmark evaluation that cross-rack transfer is indeed the most dominant factor in the overall repair performance. We study DRC(9,6,3) and DRC(9,5,3) as the representatives for Family 1 and Family 2, respectively, using the default parameters. We provide a breakdown of the repair time for a single failed block; note that the default block sizes for DRC(9,6,3) and DRC(9,5,3) are 63MiB and 64MiB, respectively. We decompose a repair operation into different steps, including sending data over the network and performing local computations in different APIs (see Section 5.2). We derive the expected running time of each step as follows:

- *Disk read*: For both DRC(9,6,3) and DRC(9,5,3), each available node first reads a block from its local disk. Our measurement indicates that the disk read throughput of our testbed is around 177MiB/s. Thus, the disk read times for a single block for DRC(9,6,3) and DRC(9,5,3) are 0.354s and 0.361s, respectively.
- *Node encode*: Each available node executes `NodeEncode` to compute an encoded subblock. Our measurement finds that the times spent on `NodeEncode` for DRC(9,6,3) and DRC(9,5,3) are 0.067s and 0.068s, which are very similar. Our investigation finds that DRC(9,6,3) only needs to perform simple node-level encoding, whereas DRC(9,5,3) does not even need to perform node-level encoding. Thus, the overhead is mainly due to the JNI calls rather than the encoding computations.
- *Inner-rack transfer*: We study the inner-rack transfer performance at the relayer in each non-local rack. Our measurement using `iperf` indicates that the effective inner-rack bandwidth of the 10Gb/s link is around 9.41Gb/s ≈ 1,090MiB/s. For DRC(9,6,3), the relayer receives an amount of $\frac{2}{3} \times 63 = 42$MiB of inner-rack traffic, so the inner-rack transfer time is 0.039s. For DRC(9,5,3), the relayer receives an amount of 64MiB of inner-rack traffic, so the inner-rack transfer time is 0.059s.
- *Relayer encode*: Each relayer executes `RelayerEncode` to re-encode the received encoded subblocks. Our measurement finds that the times spent on `RelayerEncode` for DRC(9,6,3) and DRC(9,5,3) are 0.191s and 0.145s, respectively. Although the relayer processes more input data in DRC(9,5,3) than in DRC(9,6,3), it performs simpler linear combinations (see Figure 5) and hence spends less time in `RelayerEncode`.
- *Cross-rack transfer*: We study the cross-rack transfer performance from the target's perspective. Our measurement using `iperf` indicates that when we set the gateway bandwidth (i.e., the simulated cross-rack bandwidth) to 1Gb/s, the effective bandwidth is around 953Mb/s ≈ 114MiB/s. For DRC(9,6,3), the amount of cross-rack traffic is $2 \times 63 = 126$MiB,

Table 3. Time Breakdown of Repairing a Single Failed Block (in Seconds)

	DRC(9,6,3)	DRC(9,5,3)
Disk read	0.354	0.361
Node encode	0.067	0.068
Inner-rack transfer	0.039	0.059
Relayer encode	0.191	0.145
Cross-rack transfer	1.105	0.561
Decode	0.443	0.32

so the cross-rack transfer time is 1.105s. For DRC(9,5,3), the amount of cross-rack traffic is 64MiB, so the cross-rack transfer time is 0.561s.

— *Decode*: The target executes `Decode` to obtain the reconstructed block. Our measurement indicates that the times spent on `Decode` for DRC(9,6,3) and DRC(9,5,3) are 0.443s and 0.32s, respectively.

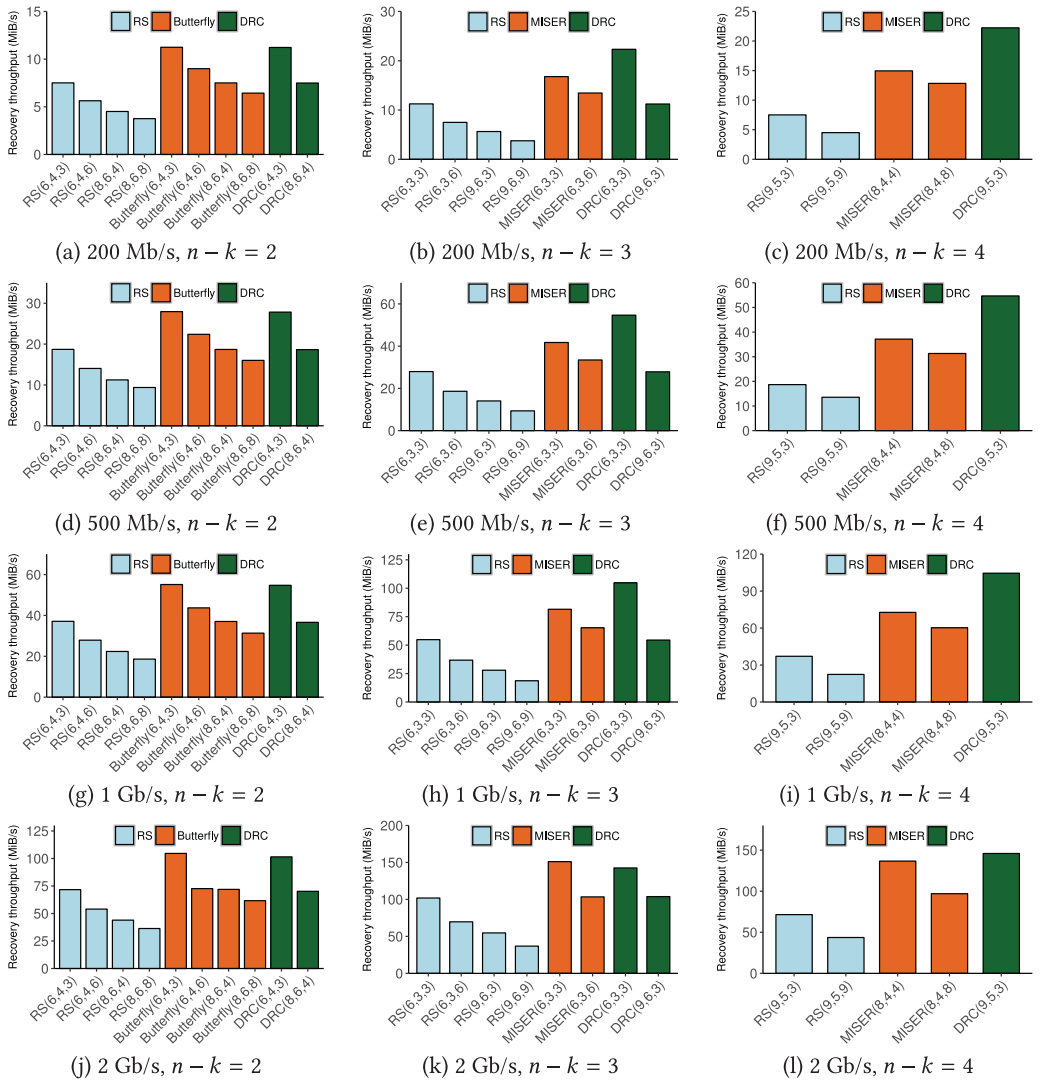
Table 3 summarizes the breakdown results. Our study shows that the cross-rack transfer time is the most dominant factor in the repair operation. If we pipeline all of the steps and run them in parallel, we expect that the repair performance is bottlenecked by the cross-rack transfer. Note that the decode time is high in both codes. Nevertheless, the actual decoding overhead can be mitigated in node recovery, as we can parallelize the decoding of multiple stripes across different targets. However, the disk read is another dominant factor in the repair performance, especially when the available cross-rack bandwidth increases. Our later experiments will further validate our microbenchmark evaluation.

6.3 Node Recovery

We first evaluate the node recovery performance of DoubleR when it repairs multiple failed blocks of a single failed node. Specifically, we write 20 stripes of blocks across DataNodes. To mimic a node failure, we pick one DataNode at random, erase all of its 20 blocks, and run DoubleR to repair all erased blocks. We use the default parameters (see Section 6.1) and vary the gateway bandwidth from 200Mb/s to 2Gb/s. For each erasure code, we measure the *recovery throughput*, defined as the total size of failed blocks being repaired divided by the total time of the entire node recovery operation.

Figure 6 shows the results for different erasure codes. When the gateway bandwidth ranges from 200Mb/s to 1Gb/s, the measured recovery throughput results are fairly consistent with the numerical results in Figure 3 in Section 3.3, as the repair performance is now bottlenecked by the available gateway bandwidth. For example, we compare RS(9,5,3) and DRC(9,5,3). From the numerical results (see Figure 3), the cross-rack repair traffic of RS(9,5,3) is three blocks for repairing a single failed block, whereas that of DRC(9,5,3) is one block only. From the measured results, the recovery throughput of DRC(9,5,3) is 2.96×, 2.92×, and 2.81× that of RS(9,5,3) when the gateway bandwidth is 200Mb/s, 500Mb/s, and 1Gb/s, respectively (see Figure 6(c), (f), and (i), respectively). Overall, when the available gateway bandwidth is smaller, the recovery throughput gain is closer to the theoretical gain.

However, when the gateway bandwidth is 2Gb/s, the disk read also becomes a dominant factor in the repair performance (see Section 6.2), so the gain of DRC diminishes. For example, the recovery throughput gain of DRC(9,5,3) over RS(9,5,3) drops to 2.04× (see Figure 6(l)). In another example, DRC(6,3,3) has 10% lower recovery throughput than MISER(6,3,3) when the gateway bandwidth



(a) 200 Mb/s, $n-k=2$ (b) 200 Mb/s, $n-k=3$ (c) 200 Mb/s, $n-k=4$

(d) 500 Mb/s, $n-k=2$ (e) 500 Mb/s, $n-k=3$ (f) 500 Mb/s, $n-k=4$

(g) 1 Gb/s, $n-k=2$ (h) 1 Gb/s, $n-k=3$ (i) 1 Gb/s, $n-k=4$

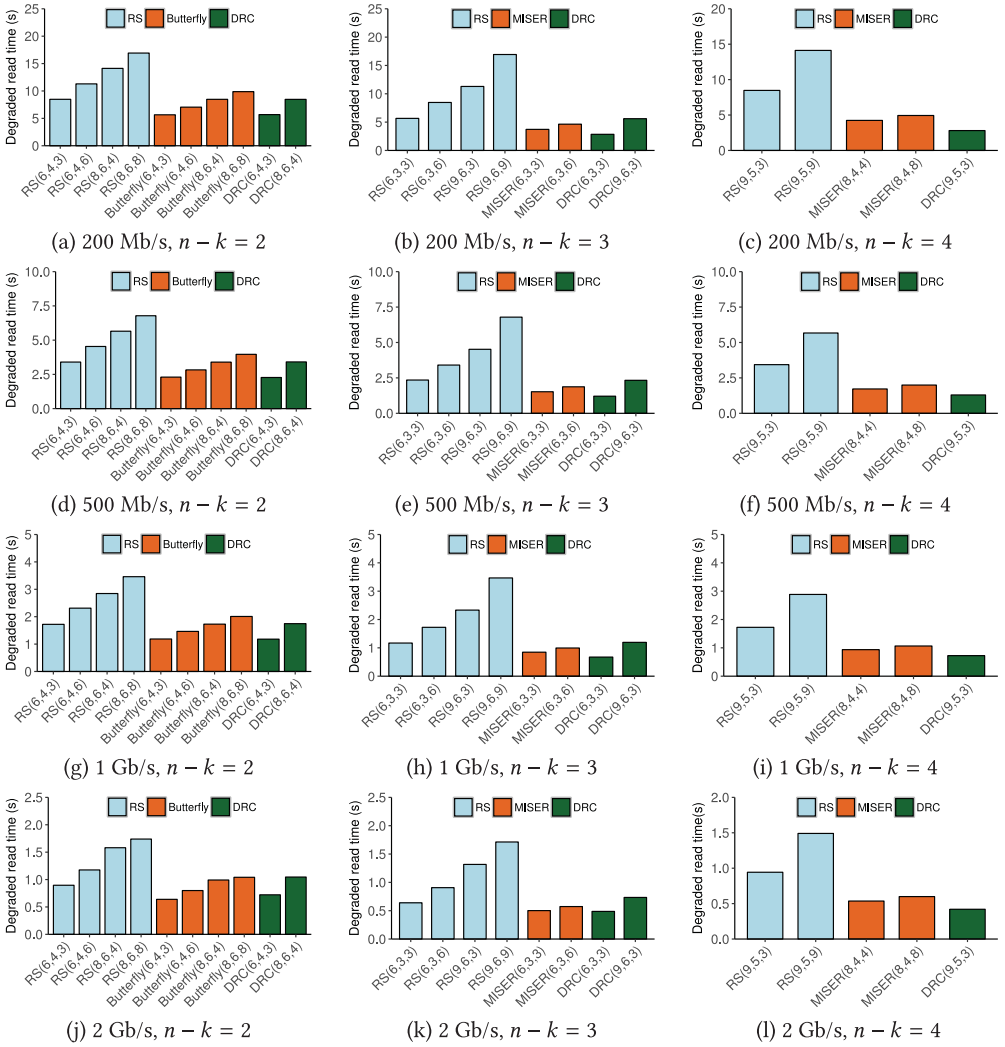
(j) 2 Gb/s, $n-k=2$ (k) 2 Gb/s, $n-k=3$ (l) 2 Gb/s, $n-k=4$

Fig. 6. Node recovery performance of different erasure codes under different gateway bandwidth settings. Note that Butterfly codes and DRC have very close performance when $r=n/2$ and $n-k=2$ (see Theorem 3.1).

is 2Gb/s (see Figure 6(k)), although it has higher throughput than MISER(6,3,3) when the gateway bandwidth is no more than 1Gb/s. Thus, we can claim the benefits of DRC only if cross-rack transfer is the performance bottleneck in a data center.

6.4 Degraded Reads

We next evaluate the degraded read performance when the file system client accesses a single unavailable block. Specifically, we randomly choose a data block to erase and let the file system client access the erased block through a degraded read. As in Section 6.3, we again use the default parameters and vary the gateway bandwidth from 200Mb/s to 2Gb/s. We measure the *degraded*



(a) 200 Mb/s, $n-k=2$ (b) 200 Mb/s, $n-k=3$ (c) 200 Mb/s, $n-k=4$

(d) 500 Mb/s, $n-k=2$ (e) 500 Mb/s, $n-k=3$ (f) 500 Mb/s, $n-k=4$

(g) 1 Gb/s, $n-k=2$ (h) 1 Gb/s, $n-k=3$ (i) 1 Gb/s, $n-k=4$

(j) 2 Gb/s, $n-k=2$ (k) 2 Gb/s, $n-k=3$ (l) 2 Gb/s, $n-k=4$

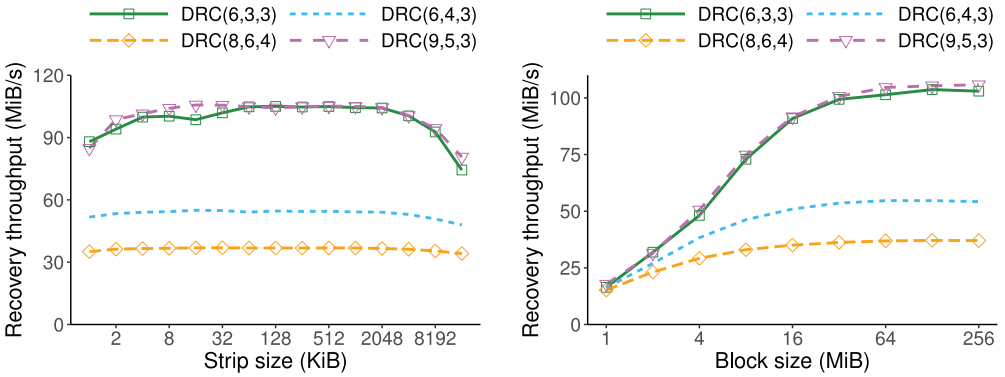
Fig. 7. Degraded read performance of different erasure codes under different gateway bandwidth settings.

read time, defined as the latency from issuing a read request until the failed block is completely reconstructed at the file system client.

Figure 7 shows the results for different erasure codes. We see that DRC also shows performance gain in degraded reads by minimizing the cross-rack repair traffic. For example, we compare RS(9,5,3) and DRC(9,5,3). The degraded read time of DRC(9,5,3) is 66.9%, 62.3%, 58.0%, and 55.4% less than that of RS(9,5,3) when the gateway bandwidth is set to 200Mb/s, 500Mb/s, 1Gb/s, and 2Gb/s, respectively.

6.5 Impact of Strip Size and Block Size

We finally evaluate the repair performance of DoubleR for various strip sizes and block sizes. Here, we focus on node recovery as in Section 6.3 and compare DRC(6,4,3), DRC(6,3,3), DRC(8,6,4), and



(a) Strip size (b) Block size

Fig. 8. Impact of strip size and block size on node recovery performance.

DRC(9,5,3). Note that we do not consider DRC(9,6,3), as its strip size and block size are different from others (see Section 6.1). We set the gateway bandwidth as 1Gb/s.

Figure 8(a) first shows the recovery throughput when the strip size varies from 1KiB to 16MiB, where the block size is fixed at 64MiB. We see that there is a performance drop when the strip size is too small or too large. Specifically, when the strip size is less than 8KiB, DoubleR needs to issue more function calls to access more strips of a block, and the overhead becomes more significant. When the strip size is larger than 2MiB, the parallelism across multiple strips of a block cannot be fully utilized. The recovery throughput is the maximum when the strip size is in between.

Figure 8(b) shows the recovery throughput when the block size varies from 1MiB to 256MiB, where the strip size is fixed at 256KiB. The recovery throughput is small when the block size is small, as the block access overhead is significant. The recovery throughput increases with the block size and reaches the maximum when the block size is at least 64MiB.

7 RELATED WORK

We review related work on erasure coding in the context of improving repair performance.

Erasure code constructions. Many constructions of erasure codes have been proposed to reduce the repair traffic. Regenerating codes [15] are a special family of erasure codes that minimize the repair traffic and provably achieve the optimal trade-off between storage redundancy and repair traffic. Constructions of regenerating codes have been proposed, such as interference alignment codes [47, 51, 55], product-matrix codes [41], zigzag codes [52], FMSR codes [9, 23], PM-RBT codes [38], and butterfly codes [35]. As stated in Section 4.1, recent studies [19, 43, 57, 58] also propose MSR code constructions for general parameters.

Some erasure codes aim to minimize I/O (i.e., the amount of data read from storage) during repair. For example, rotated RS codes [28] and Hitchhiker [40] propose new parity constructions that send fewer blocks in a single-node failure repair.

Some erasure codes trade storage efficiency for repair performance. Simple regenerating codes (SRC) [36] retrieve data from a small number of nonfailed nodes to repair a failed node, thereby limiting the I/O overhead of accessing nonfailed nodes during repair. Locally repairable codes trade storage efficiency for repair performance by associating local parity blocks with different subsets of nodes. Thus, they can retrieve data from a smaller number of nodes during repair and limit both

repair traffic and I/O. Two representative constructions are Azure's LRC [25] and Facebook's LRC [44].

The preceding erasure codes mainly adopt flat block placement in hierarchical data centers to tolerate rack failures (as mentioned in Ford et al. [16], Huang et al. [25], Muralidhar et al. [33], Rashmi et al. [40], and Sathiamoorthy et al. [44]). Our work complements the preceding studies by specifically minimizing the critical cross-rack repair traffic via hierarchical block placement.

Erasure coding in hierarchical data centers. Erasure-coded repair in hierarchical data centers has been studied, although in a limited context. Some studies focus on a data center with two racks [17, 37] or propose locally repairable codes for multiple racks [53]. R-STAIR codes [29] place an extra parity block in each rack to allow rack-local repair without cross-rack traffic. However, R-STAIR codes require sophisticated configurations of parameters of full-rack and partial-rack fault tolerance. CAR [48] is specifically designed for RS codes and exploits intra-rack encoding to reduce the cross-rack repair traffic. However, CAR does not provide any theoretical guarantee of minimizing the cross-rack repair traffic as in DRC [24]. We extend the DRC framework [24] from the applied side: we provide practical DRC constructions and evaluate their prototype implementation.

Efficient repair approaches. Some studies propose efficient repair approaches for existing erasure codes. For example, lazy repair [7, 50] triggers repair only when the number of failures reaches a threshold to avoid repairing temporary failures. CORE [31] extends existing regenerating codes to support the optimal recovery of multinode failures and presents a prototype implementation on HDFS. HACFS [56] dynamically switches encoded blocks between two erasure codes to balance storage overhead and repair performance. PPR [32] divides a repair into partial operations executed by multiple servers in parallel to reduce the overall repair time. Repair pipelining [30] further reduces the repair time to almost the same as the normal read time by slicing the repair along a linear chain. Our work differs from them by proposing new regenerating code constructions for hierarchical data centers.

8 DISCUSSION

The repair gains of DRC build on several design assumptions. In this section, we discuss the design trade-offs of DRC.

Reduced rack-level fault tolerance. DRC builds on hierarchical block placement to trade reduced rack-level fault tolerance for the minimum cross-rack repair traffic. The underlying assumption is that rack failures or correlated node failures are rare, so minimizing the cross-rack repair traffic can improve the repair performance and hence the overall storage reliability (see Section 3.4). Otherwise, erasure codes that build on flat block placement should be used.

Limited cross-rack bandwidth. DRC assumes that the repair performance is bottlenecked by the constrained cross-rack bandwidth. If the cross-rack bandwidth is sufficient, then other types of overhead may become prohibitive. For example, Family 1 of DRC needs to read $n - 1$ blocks from disk to achieve the minimum cross-rack repair traffic (same for MSR codes), whereas RS codes only need to read k blocks. In addition, although DRC minimizes the cross-rack repair traffic, its total number of blocks being transferred, including both cross rack and inner rack, is more than that of MSR codes in general (e.g., see Figure 2). Thus, the repair gain of DRC may no longer hold when the cross-rack bandwidth is sufficient, as shown in Section 6.3 for the case of 2Gb/s gateway bandwidth.

Limited parameters. Our current DRC constructions are designed for specific sets of parameters. An open question is whether there exist explicit DRC constructions for general sets of parameters.

Storage optimality. In this article, we only focus on erasure codes that are MDS, including RS codes, MSR codes, and DRC. However, if we relax the storage optimality assumption, we can further reduce or even eliminate the cross-rack repair traffic. For example, locally repairable codes [25, 44] can be deployed via hierarchical block placement by placing each local parity stripe in the same rack to eliminate the cross-rack repair traffic in a single-node repair. The trade-off of locally repairable codes is that they are non-MDS and incur higher storage redundancy than MDS codes.

9 CONCLUSIONS

We present DoubleR, a framework that realizes repair layering to improve repair performance in hierarchical data centers. DoubleR builds on the recent theoretical findings of DRC and aims to minimize the cross-rack repair traffic. We design and implement two families of practical DRC constructions for DoubleR. Experiments on our DoubleR prototype show the effectiveness of DRC in terms of node recovery throughput and degraded read time over state-of-the-art regenerating codes. The source code of our DoubleR prototype is available for download at http://adslab.cse.cuhk.edu.hk/software/doubler.

REFERENCES

[1] GitHub. 2017. Facebookarchive/hadoop-20. Retrieved October 12, 2017, from https://github.com/facebookarchive/hadoop-20.

[2] HadoopWiki. 2017. HDFS RAID. Retrieved October 12, 2017, from http://wiki.apache.org/hadoop/HDFS-RAID.

[3] GitHub. 2017. ISA-L. Retrieved October 12, 2017, from https://github.com/01org/isa-l

[4] Marcos K. Aguilera. 2013. Geo-distributed storage in data centers. Slides presented at the International Conference on Principles of Distributed Systems (OPODIS'13).

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar. 2014. ShuffleWatcher: Shuffle-aware scheduling in multi-tenant MapReduce clusters. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*. 1–12.

[6] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10)*. 267–280.

[7] Ranjita Bhagwan, Kiran Tati, Yuchung Cheng, Stefan Savage, and Geoffrey M. Voelker. 2004. Total recall: system support for automated availability management. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation (NSDI'04)*. 25.

[8] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, et al. 2011. Windows Azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. ACM, New York, NY, 143–157.

[9] Henry C. H. Chen, Yuchong Hu, Patrick P. C. Lee, and Yang Tang. 2014. NCCloud: A network-coding-based storage system in a cloud-of-clouds. *IEEE Transactions on Computers* 63, 1, 31–44.

[10] Brian Cho and Marcos K. Aguilera. 2012. Surviving congestion in geo-distributed storage systems. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 40.

[11] Mosharaf Chowdhury, Srikanth Kandula, and Ion Stoica. 2013. Leveraging endpoint flexibility in data-intensive clusters. In *Proceedings of the 2013 ACM SIGCOMM Conference (SIGCOMM'13)*. 231–242.

[12] Asaf Cidon, Robert Escriva, Sachin Katti, Mendel Rosenblum, and Emin Gün Sirer. 2015. Tiered replication: A cost-effective alternative to full cluster geo-replication. In *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. 31–43.

[13] Cisco Systems. 2016. Oversubscription and Density Best Practices. Retrieved October 12, 2017, from http://www.cisco.com/c/en/us/solutions/collateral/data-center-virtualization/storage-networking-solution/net_implementation_white_paper0900aecd800f592f.html.

[14] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)*. 10.

[15] A. G. Dimakis, P. B. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. 2010. Network coding for distributed storage systems. *IEEE Transactions on Information Theory* 56, 9, 4539–4551.

[16] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. 61–74.

[17] B. Gaston, J. Pujol, and M. Villanueva. 2013. A realistic distributed storage system that minimizes data storage and repair bandwidth. In *Proceedings of the 2013 Data Compression Conference (DCC'13)*. 491.

[18] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Vol. 37. ACM, New York, NY, 29–43.

[19] Sreechakra Goparaju, Arman Fazeli, and Alexander Vardy. 2017. Minimum storage regenerating codes for all parameters. *IEEE Transactions on Information Theory* 63, 10, 6318–6328.

[20] Kevin M. Greenan, Ethan L. Miller, and Thomas J. E. Schwarz. 2008. Optimizing Galois field arithmetic for diverse processor architectures and applications. In *Proceedings of the 2008 IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'08)*. 1–10.

[21] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. 2010. Mean time to meaningless: MTTDL, Markov models, and storage system reliability. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'10)*. 5.

[22] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. 2009. VL2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference (SIGCOMM'09)*. 51–62.

[23] Yuchong Hu, Patrick P. C. Lee, Kenneth W. Shum, and Pan Zhou. 2017. Proxy-assisted regenerating codes with uncoded repair for distributed storage systems. *IEEE Transactions on Information Theory* PP, 99, 1.

[24] Yuchong Hu, Patrick P. C. Lee, and Xiaoyang Zhang. 2016. Double regenerating codes for hierarchical data centers. In *Proceedings of the 2016 IEEE International Symposium on Information Theory (ISIT'16)*. 245–249.

[25] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. 2012. Erasure coding in Windows Azure storage. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*. 2.

[26] Virajith Jalaparti, Peter Bodík, Ishai Menache, Sriram Rao, Konstantin Makarychev, and Matthew Caesar. 2015. Network-aware scheduling for data-parallel jobs: Plan when you can. In *Proceedings of the 2015 ACM SIGCOMM Conference (SIGCOMM'15)*. 407–420.

[27] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. 2008. Are disks the dominant contributor for storage failures? A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage* 4, 3, 7.

[28] O. Khan, R. Burns, J. S. Plank, W. Pierce, and C. Huang. 2012. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*. 20.

[29] Mingqiang Li, Runhui Li, and Patrick P. C. Lee. 2016. Relieving both storage and recovery burdens in big data clusters with R-STAIR codes. *IEEE Internet Computing* PP, 99, 1.

[30] Runhui Li, Xiaolu Li, Patrick P. C. Lee, and Qun Huang. 2017. Repair pipelining for erasure-coded storage. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 567–579.

[31] Runhui Li, Jian Lin, and Patrick P. C. Lee. 2015. Enabling concurrent failure recovery for regenerating-coding-based storage systems: From theory to practice. *IEEE Transactions on Computers* 64, 7, 1898–1911.

[32] Subrata Mitra, Rajesh Panta, Moo-Ryong Ra, and Saurabh Bagchi. 2016. Partial-parallel-repair (PPR): A distributed technique for repairing erasure coded storage. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. 30.

[33] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. 2014. f4: Facebook's warm blob storage system. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI'14)*. 383–398.

[34] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. 2013. The Quantcast File System. *Proceedings of the VLDB Endowment* 6, 11, 1092–1101.

[35] Lluis Pamies-Juarez, Filip Blagojević, Robert Mateescu, Cyril Gyuot, Eyal En Gad, and Zvonimir Bandic. 2016. Opening the chrysalis: On the real repair performance of MSR codes. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies (FAST'16)*. 81–94.

[36] Dimitris S. Papailiopoulos, Jianqiang Luo, Alexandros G. Dimakis, Cheng Huang, and Jin Li. 2012. Simple regenerating codes: Network coding for cloud storage. In *Proceedings of the 2012 IEEE INFOCOM Conference*. 2801–2805.

[37] Jaume Pernas, Chau Yuen, Bernat Gastón, and Jaume Pujol. 2013. Non-homogeneous two-rack model for distributed storage systems. In *Proceedings of the 2013 IEEE International Symposium on Information Theory (ISIT'13)*.

[38] K. V. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. 2015. Having your cake and eating it too: Jointly optimal erasure codes for I/O, storage, and network-bandwidth. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 81–94.

[39] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2013. A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the

Facebook warehouse cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage'13)*. 8.

[40] K. V. Rashmi, Nihar B. Shah, Dikang Gu, Hairong Kuang, Dhruba Borthakur, and Kannan Ramchandran. 2014. A hitchhiker's guide to fast and efficient data reconstruction in erasure-coded data centers. In *Proceedings of the 2014 ACM SIGCOMM Conference (SIGCOMM'14)*. 331–342.

[41] K. V. Rashmi, Nihar B. Shah, and P. Vijay Kumar. 2011. Optimal exact-regenerating codes for distributed storage at the MSR and MBR points via a product-matrix construction. *IEEE Transactions on Information Theory* 57, 8, 5227–5239.

[42] I. S. Reed and G. Solomon. 1960. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* 8, 2, 300–304.

[43] Birenjith Sasidharan, Myna Vajha, and P. Vijay Kumar. 2016. An explicit, coupled-layer construction of a high-rate MSR code with low sub-packetization level, small field size and all-node repair. arXiv:1607.07335.

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

[45] Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST'07)*. 1.

[46] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran. 2012. Distributed storage codes with repair-by-transfer and non-achievability of interior points on the storage-bandwidth tradeoff. *IEEE Transactions on Information Theory* 58, 3, 1837–1852.

[47] N. B. Shah, K. V. Rashmi, P. V. Kumar, and K. Ramchandran. 2012. Interference alignment in regenerating codes for distributed storage: Necessity and code constructions. *IEEE Transactions on Information Theory* 58, 4, 2134–2158.

[48] Zhirong Shen, Jiwu Shu, and Patrick P. C. Lee. 2016. Reconsidering single failure recovery in clustered file systems. In *Proceedings of the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16)*. 323–334.

[49] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST'10)*. 1–10.

[50] Mark Silberstein, Lakshmi Ganesh, Yang Wang, Lorenzo Alvisi, and Mike Dahlin. 2014. Lazy means smart: Reducing repair bandwidth costs in erasure-coded distributed storage. In *Proceedings of the 2014 International Conference on Systems and Storage (SYSTOR'14)*. 1–7.

[51] C. Suh and K. Ramchandran. 2011. Exact-repair MDS code construction using interference alignment. *IEEE Transactions on Information Theory* 57, 3, 1425–1442.

[52] Itzhak Tamo, Zhiying Wang, and Jehoshua Bruck. 2013. Zigzag codes: MDS array codes with optimal rebuilding. *IEEE Transactions on Information Theory* 59, 3, 1597–1616.

[53] M. Ali Tebbi, Terence H. Chan, and Chi Wan Sung. 2014. A code design framework for multi-rack distributed storage. In *Proceedings of the 2014 IEEE Information Theory Workshop (ITW'14)*. 55–59.

[54] Amin Vahdat, Mohammad Al-Fares, Nathan Farrington, Radhika Niranjan Mysore, George Porter, and Sivasankar Radhakrishnan. 2010. Scale-out networking in the data center. *IEEE Micro* 30, 4, 29–41.

[55] Y. Wu and A. G. Dimakis. 2009. Reducing repair traffic for erasure coding-based storage via interference alignment. In *Proceedings of the 2009 IEEE International Symposium on Information Theory (ISIT'09)*. 2276–2280.

[56] Mingyuan Xia, Mohit Saxena, Mario Blaum, and David A. Pease. 2015. A tale of two erasure codes in HDFS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 213–226.

[57] Min Ye and Alexander Barg. 2017. Explicit constructions of high-rate MDS array codes with optimal repair bandwidth. *IEEE Transactions on Information Theory* 63, 4, 2001–2014.

[58] Min Ye and Alexander Barg. 2017. Explicit constructions of optimal-access MDS codes with nearly optimal sub-packetization. *IEEE Transactions on Information Theory* 63, 10, 6307–6317.

Received March 2017; revised August 2017; accepted September 2017