Reconsidering Single Failure Recovery in Clustered File Systems

Zhirong Shen†, Jiwu Shu†, and Patrick P. C. Lee‡
†Department of Computer Science and Technology, Tsinghua University
‡Department of Computer Science and Engineering, The Chinese University of Hong Kong
zhirong.shen2601@gmail.com, shujw@tsinghua.edu.cn, pclee@cse.cuhk.edu.hk
Corresponding author: Jiwu Shu (*shujw@tsinghua.edu.cn*)

***Abstract*—How to improve the performance of single failure recovery has been an active research topic because of its prevalence in large-scale storage systems. We argue that when erasure coding is deployed in a cluster file system (CFS), existing single failure recovery designs are limited in different aspects: neglecting the bandwidth diversity property in a CFS architecture, targeting specific erasure code constructions, and no special treatment on load balancing during recovery. In this paper, we reconsider the single failure recovery problem in a CFS setting, and propose CAR, a *cross-rack-aware recovery* algorithm. For each stripe, CAR finds a recovery solution that retrieves data from the minimum number of racks. It also reduces the amount of cross-rack repair traffic by performing intra-rack data aggregation prior to cross-rack transmission. Furthermore, by considering multi-stripe recovery, CAR balances the amount of cross-rack repair traffic across multiple racks. Evaluation results show that CAR can effectively reduce the amount of cross-rack repair traffic and the resulting recovery time.**

I. Introduction

Distributed computing applications often build on a clustered file system (CFS), which provides a unified storage service constructed over a number of physically independent storage servers (referred to as *nodes* in this paper). Examples of CFSes include Google File System [12], Hadoop Distributed File System [32], and Windows Azure Storage [4].

Failures are commonplace in large-scale CFSes [10], [12], [28], [29]. In particular, most failures found in CFSes are *single node failures* (or single failures in short), which can account for over 90% of all failure events in real deployment [10]. To maintain data availability in the presence of failures, a common approach is to store data with redundancy. Compared with traditional replication, *erasure coding* is shown to achieve higher fault tolerance with less redundancy [33], and hence is increasingly used in today's CFSes for improved storage efficiency. Mainstream erasure codes work by taking original pieces of information (termed *data chunks*) as inputs and produce additional redundant pieces of information (termed *parity chunks*), such that if any data or parity chunk is lost, we can retrieve other available data and parity chunks to reconstruct the lost chunk. The collection of data and parity chunks that are encoded together is called a *stripe*. A CFS stores multiple stripes of information, each of which is independently encoded/decoded according to the erasure code constructions.

Given the prevalence of single failures, there have been a spate of solutions (e.g., [11], [16], [24], [25], [31], [36], [38], [39]) on improving the performance of single failure recovery in erasure-coded storage systems. The main idea of such solutions is to selectively retrieve different portions of data and parity chunks within a stripe, with a common objective of minimizing the amount of *repair traffic* (i.e., the amount of information retrieved from surviving nodes for data reconstruction). On the other hand, when we examine existing single failure recovery solutions, there remain three limitations.

First, existing studies on single failure recovery neglect the *bandwidth diversity* property in a CFS architecture. Typical CFS architectures organize nodes in multiple racks, in which intra-rack and cross-rack connections have different bandwidth capacities. In practice, intra-rack bandwidth is considered to be sufficient, while cross-rack bandwidth is often over-subscribed and considered to be a scarce resource [6], [8], [18]. The lack of considerations on bandwidth diversity may lead to inefficient recovery solutions that trigger massive cross-rack data transmissions, thereby degrading the overall recovery performance.

Second, many single failure recovery solutions (e.g., [11], [36], [38], [39]) focus on XOR-based erasure codes, which are not commonly used for maintaining fault tolerance in a CFS. XOR-based erasure codes (e.g., [2], [7], [14], [15], [30], [34], [37]) are special classes of erasure codes that perform encoding/decoding via XOR operations only, thereby achieving high encoding/decoding performance. However, XOR-based erasure codes often target specific fault tolerance settings (e.g., RDP codes [7] are RAID-6 codes that are double-fault-tolerant). In view of generality and flexibility, today's CFSes (e.g., [1], [10], [21]) usually employ Reed-Solomon (RS) codes [26] for general fault tolerance. RS codes perform encoding/decoding operations over finite fields [23], and have inherently different constructions from XOR-based erasure codes. It is non-trivial to apply existing single failure recovery solutions that are designed for XOR-based erasure codes directly to RS codes.

Third, existing single failure recovery solutions focus on minimizing the amount of repair traffic, but most of them do not consider the load balancing issue during the recovery operation. It is possible that the recovery performance is bottlenecked by one or few nodes that send more repair traffic than others.

To address the above limitations, we reconsider the single failure recovery problem in a CFS setting. First, we should specifically minimize the amount of *cross-rack repair traffic* (i.e., the amount of data to be retrieved across racks for data reconstruction), which plays an important role in improving the single failure recovery performance with regard to the scarce cross-rack bandwidth in a CFS. Second, our single failure recovery design should address general fault tolerance (e.g., based on RS codes). Finally, we should balance the amount of cross-rack repair traffic at the rack level (i.e., across multiple racks) while keeping the total amount of cross-rack repair traffic minimum, so as to ensure that the single failure recovery performance is not bottlenecked by a single rack.

To this end, we propose *cross-rack-aware recovery* (CAR), a new single failure recovery algorithm that aims to reduce and balance the amount of cross-rack repair traffic for a single failure recovery in a CFS that deploys RS codes for general fault tolerance. CAR has three key techniques. First, for each stripe, CAR examines the data layout and finds a recovery solution in which the resulting repair traffic comes from the minimum number of racks. Second, CAR performs intra-rack aggregation for the retrieved chunks in each rack before transmitting them across racks in order to minimize the amount of cross-rack repair traffic. Third, CAR examines the per-stripe recovery solutions across multiple stripes, and constructs a multi-stripe recovery solution that balances the amount of cross-rack repair traffic across multiple racks.

Our contributions are summarized as follows.

- We reconsider the single failure recovery problem in the CFS setting, and identify the open issues that are not addressed by existing studies on single failure recovery.
- We propose CAR, a new cross-rack-aware single failure recovery algorithm for a CFS setting. CAR is designed based on RS codes. It reduces the amount of cross-rack repair traffic for each stripe, and additionally searches for a multi-stripe recovery solution that balances the amount of cross-rack repair traffic across racks.
- We implement CAR and conduct extensive testbed experiments based on different CFS settings with up to 20 nodes. We show that CAR can reduce 66.9% of cross-rack repair traffic and 53.8% of recovery time when compared to a baseline single failure recovery design that does not consider the bandwidth diversity property of a CFS. Also, we show that CAR effectively balances the amount cross-rack repair traffic across racks.

The rest of this paper proceeds as follows. Section II presents the background details of erasure coding and reviews related work on single failure recovery. Section III formulates and motivates the problem in the CFS setting. Section IV presents the design of CAR. Section V presents our evaluation results on CAR based on testbed experiments. Finally, Section VI concludes the paper.

II. Background and Related Work

In this section, we provide the background details on erasure coding in the context of a CFS. We also present the open issues to be addressed in this paper.

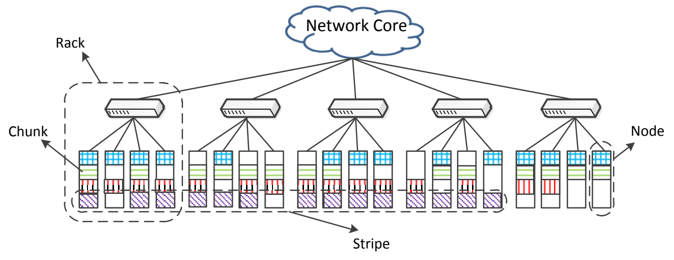


Fig. 1. Illustration of a CFS architecture that is composed of five racks with four nodes each. The CFS contains four stripes of 14 chunks encoded by a $(k = 8, m = 6)$ code, in which the chunks with the same color and fill pattern belong to the same stripe. Note that the number of chunks in each node may be different.

A. Basics

This paper considers a special type of distributed storage system architecture called a *clustered file system (CFS)*, which arranges storage nodes into racks, such that all nodes within the same rack are connected by a top-of-rack switch, while all racks are connected by a network core. Figure 1 illustrates a CFS composed of five racks with four nodes each (i.e., 20 nodes in total). Some well-known distributed storage systems, such as Google File System [12], Hadoop Distributed File System [32], and Windows Azure Storage [4], realize the CFS architecture.

We use erasure coding to maintain data availability for a CFS. We consider a popular family of erasure codes that are: (1) *Maximum Distance Separable (MDS)* codes, meaning that fault tolerance is achievable with the minimum storage redundancy (i.e., the optimal storage efficiency), and (2) *systematic*, meaning that the original data is retained after encoding. Specifically, we construct a (k, m) code (which is MDS and systematic) with two configurable parameters k and m. A (k, m) code takes k original (uncoded) data chunks of the same size as inputs and produces m (coded) parity chunks that are also of the same size, such that any k out of the $k + m$ chunks can sufficiently reconstruct all original data chunks. The $k+m$ chunks collectively form a *stripe*, and are distributed over $k+m$ different nodes. Note that the placement of chunks should also ensure rack-level fault tolerance [18], such that there are at least k chunks for data reconstruction in other surviving racks in the presence of rack failures. We address this issue when we design CAR (see Section IV).

For an erasure-coded CFS that stores a large amount of data, it contains multiple independent stripes of data/parity chunks. In this case, each node stores a different number of chunks that belong to multiple stripes. For example, referring to the CFS in Figure 1, there are four stripes spanning over 20 nodes, in which the leftmost node stores four chunks, while the rightmost node stores only two chunks.

B. Erasure Code Constructions

There have been various proposals on erasure code construction in the literature. Practical erasure codes often realize

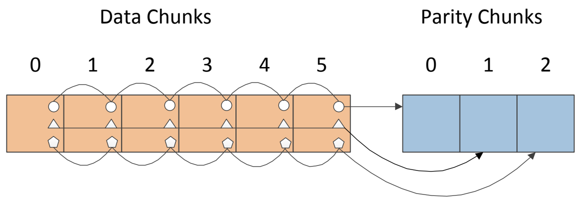


Fig. 2. Encoding of the ($k = 6, m = 3$) RS codes for a stripe, in which there are six data chunks and three parity chunks. If one of the chunks (either a data chunk or a parity chunk) fails, any six surviving chunks within the stripe can be retrieved for data reconstruction.

encoding/decoding operations based on the arithmetic over the Galois field [23]. Reed Solomon (RS) codes [26] are one representative example. RS codes are MDS, and support any pair of (k, m) in general. For example, Figure 2 shows a stripe of the ($k = 6$, $m = 3$) RS code, which contains six data chunks and three parity chunks (i.e., $m = 3$). RS codes have been intensively used for erasure-coded storage in today's commercial storage systems for fault tolerance, such as Google's ColossusFS [1] and Facebook's HDFS [3]. In this paper, we design our CAR based on RS codes.

XOR-based erasure codes are a special family of erasure codes that perform encoding/decoding with XOR operations only. Examples of XOR-based erasure codes include RDP Code [7], X-Code [37], STAR Code [14], P-Code [15], HDP Code [34], and HV Code [30]. XOR-based erasure codes are generally MDS, but they often have specific restrictions on the parameters k and m. For example, RDP Code [7] requires $(k = p-1, m = 2)$, X-Code [37] requires $(k = p-2, m = 2)$, and STAR Code [14] requires $(k = p, m = 3)$, where p is a prime number. Thus, XOR-based erasure codes are mainly used in local disk arrays.

Single failures (e.g., a single node failure or a single lost chunk within a stripe) are known to be the most common failure events in a CFS [10], [13]. In RS codes, k chunks are needed to be retrieved to recover a single lost chunk. Some erasure codes are specially designed for improving the performance of recovering a single failure. Regenerating codes [9] minimize the amount of repair traffic by allowing other surviving nodes to send computed data for data reconstruction, and achieve the optimal tradeoff between the level of storage redundancy and the amount of repair traffic. In particular, minimum-storage regenerating (MSR) codes [9] are MDS, and they minimize the amount of repair traffic subject to the minimum storage redundancy. Rashmi et al. [24] propose a new MSR code construction that also minimizes the amount of I/Os. Huang et al. [13] and Sathiamoorthy et al. [27] develop local reconstruction codes to reduce the amount of repair traffic, while incurring slightly more storage redundancy (and hence the codes are non-MDS).

Recent erasure codes address mixed failures (e.g., a combination of disk failures and sector errors) in a storage efficient way. Examples are SD codes [22] and STAIR Codes [17]. They are non-MDS, and perform encoding/decoding operations over the Galois field. They are mainly designed for local disk arrays.

C. Single Failure Recovery

There have been extensive studies in the literature that focus on improving the performance of single failure recovery. In addition to new erasure code constructions such as regenerating codes and local reconstruction codes (see Section II-B), previous studies (e.g., [11], [16], [20], [36], [39]) pay close attention to XOR-based erasure codes. To reconstruct a lost chunk, the core idea of their proposals is to examine the relationship between the data and parity chunks of a stripe and then read different portions of a stripe, so as to minimize the amount of I/Os to access the storage nodes, and hence the amount of repair traffic, in single failure recovery. Some previous studies target specific XOR-based erasure code constructions. For example, Xiang et al. [36] and Xu et al. [38] prove the theoretical lower bound on the amount of I/Os for a single failure recovery for RDP Code and X-Code, respectively, both of which tolerate two node failures.

Some previous studies focus on minimizing the amount of I/Os for single failure recovery for general XOR-based erasure codes. Khan et al. [16] propose to enumerate all possible single failure recovery solutions and select the one that minimizes the amount of I/Os. Luo et al. [20] and Fu et al. [11] extend the enumeration approach of Khan et al. [16] to load-balance the amount of I/Os to be read from surviving disks. Note that the enumeration approach is generally NP-hard. Thus, Zhu et al. [39] and Shen et al. [31] propose a greedy algorithm search for the single failure recovery solution with the near-minimum amount of I/Os, while still supporting general XOR-based erasure codes.

D. Other Problems

Some studies also study the performance issues when deploying erasure coding in a CFS. Chan et al. [5] propose a new CFS called CodFS that mitigates the parity update overhead under update-intensive workloads through an enhanced parity logging approach.

Li et al. [18] propose an efficient replica placement algorithm in a CFS to reduce the amount of cross-rack traffic when transforming replicated data to erasure-coded data. Li et al. [19] focus on improving MapReduce performance on erasure-coded storage, and propose to give degraded MapReduce tasks (i.e., the tasks that need to reconstruct and process unavailable data chunks) a higher execution priority so as to mitigate network resource competitions. Xia et al. [35] present a new approach to switch between two erasure codes to balance the storage overhead and recovery performance.

E. Open Issues

In summary, there have been extensive studies on improving the performance of single failure recovery when deploying erasure coding in disk arrays or CFSes. On the other hand, we identify three open issues that are still unexplored when reconsidering the single failure recovery problem a CFS setting.

We have provided an overview of the open issues in Section I, and following discussion provides more detailed explanations.

Lack of considerations on cross-rack repair traffic. Existing single failure recovery optimizations [11], [16], [20], [31], [36], [38], [40], while significantly reducing the amount repair traffic, do not differentiate intra-rack and cross-rack data transmissions during recovery. In particular, a CFS architecture exhibits the property of *bandwidth diversity*, in which cross-rack bandwidth is often much more limited than intra-rack bandwidth, and it is considered as an over-subscribed, scarce resource in a CFS [6], [8], [18]. A recovery solution that triggers a large amount of cross-rack traffic will unavoidably delay data reconstruction. How to minimize the amount of cross-rack repair traffic (i.e., the amount of data traffic triggered during recovery) should be carefully studied in a CFS setting.

Ineffectiveness for RS codes. Most existing studies [11], [16], [20], [31], [36], [38], [40] on single failure recovery mainly focus on XOR-based erasure codes. However, their approaches cannot be easily generalized for RS codes, which are commonly deployed in today's CFSes for general fault tolerance [1], [10]. In general, RS codes reconstruct a lost chunk by retrieving any k surviving chunks within the same stripe. This strategies imply that there are a maximum of $C(k+m-1,k)$ possible single failure recovery solutions (i.e., the number of combinations of selecting k out of $(k+m-1)$ surviving chunks). Furthermore, how to select the one with the minimum cross-rack repair traffic remains unexplored.

Load balancing of cross-rack repair traffic in a multi-stripe setting. Recall that a CFS often organizes data in multiple stripes, each of which is independently encoded (see Section II-A). Existing single failure recovery solutions mainly focus on a single stripe. It is possible to further improve load balancing of a single failure recovery if we can consider a multi-stripe setting [11], [31], [38]. However, the load balancing schemes [11], [31], [38] only target XOR-based erasure codes, and also do not address the bandwidth diversity issue in a CFS.

In a CFS, we are interested in balancing the amount of cross-rack repair traffic across multiple racks. However, solving single failure recovery problem for RS codes in a multi-stripe setting is non-trivial. As discussed above, a single failure recovery solution for a single stripe has a maximum of $C(k+m-1,k)$ possible options. If we consider $s>1$ stripes, then the total number of possible options will increase to $C(k+m-1,k)^s$. How to efficiently search for a multi-stripe single failure recovery solution will be critical.

III. Problem Formulation

This paper aims to address the following problem: *Given a CFS that deploys RS codes, can we simultaneously minimize and balance the amount of cross-rack repair traffic when we perform single failure recovery in the CFS?* In this section, we formulate the single failure recovery problem in a CFS setting. Table I summarizes the major notation used in this paper.

TABLE I
Major notation used in this paper.

Notation	Description
k	number of data chunks in a stripe
m	number of parity chunks in a stripe
r	number of racks in a CFS
s	number of stripes associated with the lost chunks
A_i	the i-th $(1 \le i \le r)$ rack
A_f	the rack where the failed node resides $(1 \le f \le r)$
λ	load balancing rate
$t_{i,f}$	cross-rack traffic on A_i to repair a failed node in A_f
$c_{i,j}$	number of chunks of the j-th stripe in rack A_i
H_i	the i-th chunk
H'_i	the i-th retrieved chunk for data reconstruction
e	number of iterations in the greedy algorithm for load balancing

Consider a CFS that deploys a (k,m) RS code over r racks denoted by $\{A_1, A_2, \cdots, A_r\}$. Suppose that a node fails, and we need to reconstruct the lost chunks in the failed node. Each stripe contains exactly one lost chunk. To make our analysis general, we assume that the lost chunks to be reconstructed in the failed node span $s \ge 1$ stripes. We denote the rack that contains the failed node as A_f $(1 \le f \le r)$; also, we call the remaining racks (aside A_f) to be *intact racks* since the data stored in all their nodes remains intact. To repair the lost chunks in the failed node, suppose the cross-rack repair traffic triggered from rack A_i is $t_{i,f}$ $(1 \le i \ne f \le r)$. We define the *load balancing rate* λ as the ratio of the maximum amount of cross-rack repair traffic across each rack to the average amount of cross-rack repair traffic over the $(r-1)$ intact racks.

$$\lambda = \frac{\max\{t_{i,f} | 1 \le i \ne f \le r\}}{\sum\limits_{1 \le i \ne f \le r} \frac{t_{i,f}}{r-1}}.$$

Obviously, if there exists cross-rack repair traffic, then $\lambda \ge 1$. Also, we say that the recovery solution is more balanced if its load balancing rate is closer to 1. Therefore, we can formulate the following optimization problem:

$$\text{Minimize} \quad \lambda$$

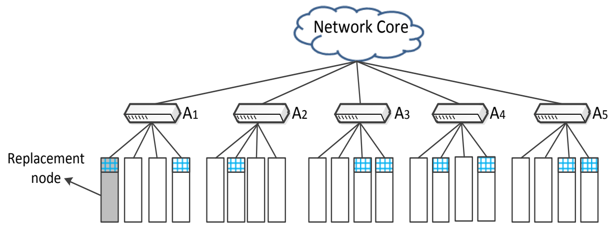
subject to

$$\sum_{1 \le i \ne f \le r} t_{i,f} \text{ is minimized.}$$

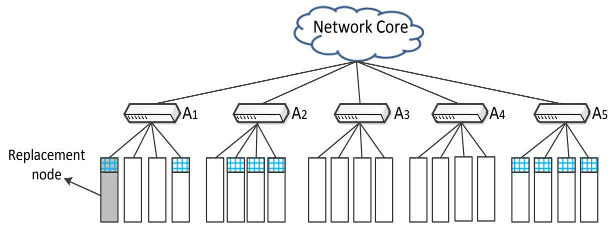
Our optimization goal is to minimize the load balancing rate, subject to the condition that the total amount of cross-rack repair traffic is minimized.

IV. Cross-rack-aware Recovery

We present CAR, a *cross-rack-aware recovery* algorithm that aims to solve the optimization problem in Section III. We focus on single failure recovery, in which there is a single failed node and each stripe contains exactly one lost chunk.



(a) Recovery that retrieves chunks from five racks.



(b) Recovery that retrieves chunks from three racks.

Fig. 3. Two recovery solutions that retrieve data from different sets of racks. Suppose that intra-rack chunk aggregation is performed. To reconstruct the lost chunk of a stripe, for (a), four chunks are transmitted across racks, while for (b), only two chunks are transmitted across racks.

A. Overview

CAR has three design objectives.

- For each stripe, finding a recovery solution that retrieves chunks from the minimum number of racks.
- Exploiting intra-rack chunk aggregation.
- Exploiting a greedy approach to search for a load-balanced multi-stripe recovery solution.

We justify the design objectives as follows. For each stripe constructed by a (k, m) RS code, any k chunks are sufficient to reconstruct the lost chunk in the stripe. Here, we examine the placement of chunks across racks and choose a recovery solution identify a recovery solution that retrieves chunks from the minimum number of racks. To repair the lost chunk, instead of directly retrieving and sending individual chunks from a rack, we perform intra-rack chunk aggregation on the retrieved chunks in the same rack and send *one* aggregated chunk (which has the same size as each data/parity chunk) to the replacement node for data reconstruction. Intra-rack chunk aggregation can be realized by separating the reconstruction process of RS codes. By retrieving chunks from the minimum number of racks and performing intra-rack chunk aggregation, we minimize the amount of cross-rack repair traffic to reconstruct the lost chunk for each stripe.

For example, suppose that the first node fails in the CFS shown in Figure 1, which adopts the $(k = 8, m = 6)$ RS code for fault tolerance. Figure 3 presents two possible recovery solutions, both of which retrieve $k = 8$ chunks yet from a different set of racks to reconstruct the lost chunk of a stripe. By performing intra-rack chunk aggregation, the requested chunks within the same rack will be aggregated into a single chunk. Therefore, the recovery solution in Figure 3(a) transmits four chunks across racks (i.e., from A_2, A_3, A_4, and A_5), while the one in Figure 3(b) only needs to transmit two chunks across racks (i.e., from A_2 and A_5). Note that the retrieval of chunks in A_1 only triggers intra-rack data transmissions, and we assume that it brings limited overhead to the overall recovery performance in a CFS.

In addition, we examine the per-stripe recovery solutions across multiple stripes so as to minimize the load balancing rate. We propose a greedy algorithm that can search for a near-optimal solution with low computational complexity.

B. Minimizing the Number of Accessed Racks

We first study how to find a recovery solution that retrieves chunks from the minimum number of racks. Suppose that the lost chunks span s stripes. For the j-th stripe $(1 \leq j \leq s)$, let $c_{i,j}$ be the number of chunks stored in the i-th rack A_i $(1 \leq i \leq r)$. Note that we also ensure that the placement of chunks provides rack-level fault tolerance [18]. Here, we assume that we provide single-rack fault tolerance. For the (k, m) RS code, we require that $c_{i,j} \leq m$, so as to tolerate any single-rack failure; in other words, each stripe should contain at least k chunks in other intact racks of the CFS for data reconstruction.

Suppose that a node fails in rack A_f $(1 \leq f \leq r)$. We use $c'_{f,j}$ to denote the number of surviving chunks of the j-th stripe $(1 \leq j \leq s)$ in A_f in the presence of the node failure. Since every node keeps at most one chunk for a given stripe, we have the following equation:

$$c'_{f,j} = \begin{cases} c_{f,j}, & \text{if} \quad c_{f,j} = 0 \\ c_{f,j} - 1, & \text{if} \quad c_{f,j} \neq 0 \end{cases} \tag{1}$$

Meanwhile, for the remaining $r - 1$ intact racks (i.e., $\{A_1, \cdots, A_{f-1}, A_{f+1}, \cdots, A_r\}$), they still have the same numbers of chunks in the j-th stripe (i.e., $\{c_{1,j}, \cdots, c_{f-1,j}, c_{f+1,j}, \cdots, c_{r,j}\}$). Given this new setting, Theorem 1 states how to determine the minimum number of intact racks to be accessed when recovering the lost chunk in the j-th stripe $(1 \leq j \leq s)$.

Theorem 1. *For the j-th stripe $(1 \leq j \leq s)$, suppose that the numbers of chunks in the $r - 1$ intact racks are ranked in descending order denoted by $\{c_{j_1}, c_{j_2}, \cdots, c_{j_{r-1}}\}$, where $c_{j_1} \geq c_{j_2} \geq \cdots \geq c_{j_{r-1}}$. We find the smallest number d_j that satisfies:*

$$c_{j_1} + \cdots + c_{j_{d_j}} + c'_{f,j} \geq k. \tag{2}$$

Then d_j is the minimum number of intact racks to be contacted to recover the lost chunk in the j-th stripe.

Proof (Sketch): We prove by contradiction. Suppose that d_j is not the minimum number of intact racks. Let $d'_j < d_j$ be the minimum number of intact racks to be accessed. Then we must have $c_{j_1} + \cdots + c_{j_{d'_j}} + c'_{f,j} \geq k$ so that the lost chunk

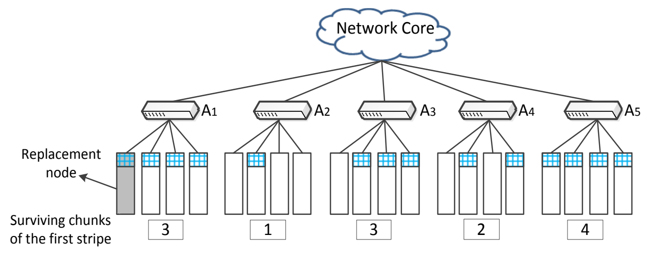


Fig. 4. Example of determining the minimum number of intact racks to be accessed when recovering the lost chunk in the first stripe. Suppose that the CFS employs the $(k = 8, m = 6)$ RS code, and that the first node in A_1 fails. The replacement node can retrieve chunks from the intact racks A_3 and A_5, as well as from the nodes within the same rack A_1, for data reconstruction.

in the j-th stripe can be reconstructed. However, this violates our condition that d_j is the minimum value for Equation (2) to be satisfied. $\square$

We elaborate Theorem 1 via an example. Consider the recovery for the first stripe in the CFS in Figure 4. The CFS has five racks and employs the $(k = 8, m = 6)$ RS code. For the first stripe, the first rack A_1 originally keeps $c_{1,1} = 4$ chunks. Suppose that the first node in A_1 fails. Then there are $c'_{1,1} = c_{1,1} - 1 = 3$ surviving chunks in A_1. The numbers of surviving chunks in other four intact racks A_2, A_3, A_4 and A_5 are $c_{2,1} = 1$, $c_{3,1} = 3$, $c_{4,1} = 2$, and $c_{5,1} = 4$, respectively. To reconstruct the lost chunk, we need $k = 8$ surviving chunks for the reconstruction in RS codes. To determine the minimum number of intact racks to be accessed, we first sort the numbers of surviving chunks in the four intact racks, and obtain $(4, 3, 2, 1)$. We can then find $d_1 = 2$, since $4 + 3 + c'_{1,1} = 10 > k = 8$. Thus, we should retrieve the surviving chunks from A_5 and A_3, as well as the surviving chunks in A_1, to reconstruct the lost chunk.

We say that a recovery solution is *valid* if it can recover the lost chunk for the j-th stripe $(1 \le j \le s)$ by accessing d_j intact racks only. A valid solution of the j-th stripe $(1 \le j \le s)$ should satisfy the condition that the number of retrieved chunks from d_j intact racks plus the number of surviving chunks in A_f should be no less than k.

We emphasize that a stripe may contain more than one valid recovery solution. We again consider the example of Figure 4. In addition to the recovery solution that retrieves surviving chunks from A_3 and A_5, we can also find another recovery solution that retrieves chunks from A_3 and A_4 instead, since $c_{3,1} + c_{4,1} + c'_{1,1} = k = 8$. The latter recovery solution is also valid, since it can also repair the lost chunk by accessing $d_1 = 2$ intact racks only.

C. Intra-rack Chunk Aggregation

After finding the minimum number of intact racks to be accessed for recovery, we perform intra-rack chunk aggregation for the retrieved chunks in the same rack. We call the aggregation operation *partial decoding*, since it performs part of the decoding steps to reconstruct the lost chunk of a stripe.

To describe how partial decoding works, we first review the encoding and decoding procedures of the (k, m) RS code. Suppose there are k data chunks $\{H_1, H_2, \cdots, H_k\}$. Note that most practical storage systems deploy systematic erasure codes (see Section II-A), meaning that the original data chunks are kept in uncoded form after encoding and hence read requests can directly access the original data. To generate the m parity chunks (denoted by $\{H_{k+1}, \cdots, H_{k+m}\}$), the encoding operation can be realized by multiplying a $(k+m) \times k$ matrix $\mathcal{G} = \big(\mathbf{g}_1 \cdots \mathbf{g}_{k+m}\big)^T$ with the k data chunks, i.e.,

$$\begin{pmatrix} \mathbf{g}_1 \\ \vdots \\ \mathbf{g}_k \\ \vdots \\ \mathbf{g}_{k+m} \end{pmatrix} \cdot \begin{pmatrix} H_1 \\ \vdots \\ H_k \end{pmatrix} = \begin{pmatrix} H_1 \\ \vdots \\ H_k \\ \vdots \\ H_{k+m} \end{pmatrix} \tag{3}$$

Here, $\mathbf{g}_i$ $(1 \le i \le k+m)$ is a row vector and its size is $1 \times k$. To make the original data kept in uncoded form, $\big(\mathbf{g}_1 \cdots \mathbf{g}_k\big)^T$ should be a $k \times k$ identity matrix, where T denotes a matrix or vector transpose operation.

In the decoding operation, RS codes can always use any k surviving chunks (denoted by $\{H'_1, \cdots, H'_k\}$) to reconstruct the original data chunks. This implies that there always exists a $k \times k$ invertible matrix $\mathcal{X}$, such that

$$\mathcal{X} \cdot \begin{pmatrix} H'_1 \\ \vdots \\ H'_k \end{pmatrix} = \begin{pmatrix} H_1 \\ \vdots \\ H_k \end{pmatrix} \tag{4}$$

Therefore, to reconstruct a chunk H_i $(1 \le i \le k+m)$, we can derive the following equation based on Equations (3) and (4).

$$H_i = \mathbf{g}_i \cdot \begin{pmatrix} H_1 \\ \vdots \\ H_k \end{pmatrix} = \mathbf{g}_i \cdot \mathcal{X} \cdot \begin{pmatrix} H'_1 \\ \vdots \\ H'_k \end{pmatrix} \tag{5}$$

Let $\mathbf{y} = \mathbf{g}_i \cdot \mathcal{X}$. As the sizes of $\mathbf{g}_i$ and $\mathcal{X}$ are $1 \times k$ and $k \times k$, respectively, $\mathbf{y} = \big(y_1 \cdots y_k\big)$ is a $1 \times k$ vector. Then we can derive the following equation based on Equation (5).

$$H_i = \mathbf{y}_i \cdot \begin{pmatrix} H'_1 \\ \vdots \\ H'_k \end{pmatrix} = \big(y_1 \cdots y_k\big) \cdot \begin{pmatrix} H'_1 \\ \vdots \\ H'_k \end{pmatrix} \tag{6}$$

Equation (6) implies that the reconstruction of H_i is actually realized by the linear operations performed on the k retrieved chunks. Therefore, to mitigate the cross-rack data transmissions for recovery, we can "aggregate" the retrieved chunks in the same rack before performing cross-rack data transmissions. For example, without loss of generality, suppose that the first j requested chunks $\{H'_1, \cdots, H'_j\}$ are stored in the same rack. Then we can specify a node in that rack to perform the linear operations based on Equation (6) and obtain the following result:

$$\sum_{i=1}^{j} y_i H'_i \tag{7}$$

Algorithm 1: Reconstruction for a stripe.

```
Input: The set of requested chunks $\{H'_1, \cdots, H'_k\}$ for
       recovering the lost chunk of a stripe.
1 for each rack do
2     if this rack stores requested chunks then
3         Specify a node in this rack to retrieve the
          requested chunks
4         Perform partial decoding on the requested chunks
5         Send the partially decoded chunk to the
          replacement node
6 Add the received partially decoded chunks at the
  replacement node to recover the lost chunk.
```

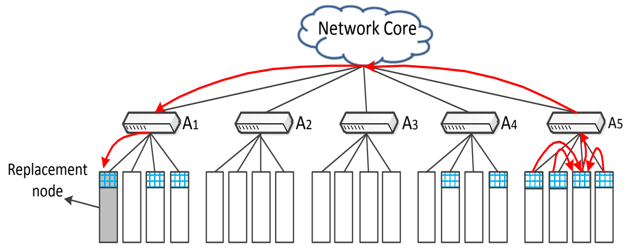


Fig. 5. Example of reconstructing the lost chunk in the first stripe via partial decoding. For example, four chunks in rack A_5 are selected for reconstruction. One node in A_5 performs partial decoding on the four selected chunks and sends the partially decoded chunk to the replacement node.

The aggregation in Equation (7) is called *partial decoding* and the output is referred to as the *partially decoded chunk*, which has the identical size as each data/parity chunk. The partially decoded chunk will then be sent to the replacement node to complete the reconstruction of the lost chunk. The replacement node simply adds all the partially decoded chunks received from A_f and other intact racks that are accessed, in order to reconstruct the lost chunk. We can observe that after applying partial decoding, the amount of cross-rack repair traffic per stripe in CAR is equal to the number of partially decoded chunks transmitted from the accessed intact racks, or equivalently, the number of intact racks to be accessed for recovery. Algorithm 1 summarizes the details of recovering the lost chunk of a stripe.

Figure 5 shows an example of how we reconstruct the lost chunk of a stripe via partial decoding. Suppose that we need to retrieve $k = 8$ chunks, and the requested chunks are denoted by $\{H'_1, H'_2, \cdots, H'_8\}$ (from left to right). To recover the lost chunk in rack A_1, we first perform the partial decoding by aggregating the requested chunks in A_1, A_4, and A_5 to be $\sum_{i=1}^{2} y'_i H'_i$, $\sum_{i=3}^{4} y'_i H'_i$, and $\sum_{i=5}^{8} y_i H'_i$, respectively. After that, the replacement node reads the three partially decoded chunks to reconstruct the lost chunk. In this example, there are only two chunks transmitted across racks.

Algorithm 2: Greedy algorithm for load balancing.

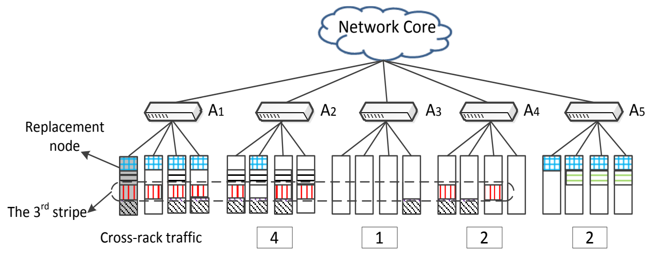
```
Input: Number of iterations $e$; number of stripes $s$
Output: A multi-stripe recovery solution $\mathbb{R}$.
1 for $j = 1$ to $s$ do
2     Select a valid recovery solution $R_j$ for the $j$-th stripe
3 Initialize $\mathbb{R} = \{R_1, R_2, \cdots, R_s\}$
4 for iteration 1 to $e$ do
5     Find intact rack $A_l$ $(1 \leq l \neq f \leq r)$ with the highest
      $t_{l,f}$
6     for each intact rack $A_i$ $(1 \leq i \neq f \leq r)$ do
7         if $A_i \neq A_l$ and $t_{l,f} - t_{i,f} \geq 2$ then
8             Find $R_j$ and another valid recovery solution
              $R'_j$ that retrieves no data from $A_l$ but from
              $A_i$ instead
9             if both $R_j$ and $R'_j$ exist then
10                Set
                  $\mathbb{R} = \{R_1, \cdots, R_{j-1}, R'_j, R_{j+1}, \cdots, R_s\}$
11                Jump to the next iteration of the for-loop
                  in step 4
12    Exit the for-loop in step 4 if there is no substitution
      in $\mathbb{R}$
```

D. Load Balancing

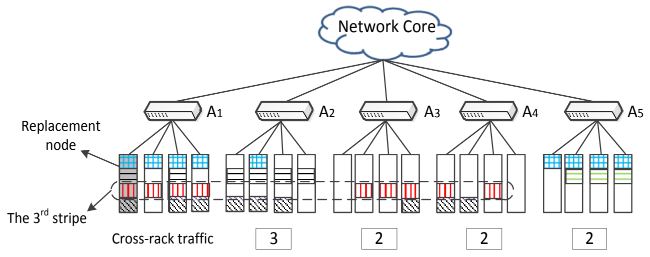
As stated in Section IV-B, each stripe can have multiple valid per-stripe recovery solutions. Here, we examine the valid per-stripe recovery solutions across multiple stripes, so as to balance the amount of cross-rack repair traffic across the racks (i.e., minimizing the load balancing rate in Section III). However, enumerating all possible valid per-stripe recovery solutions can be expensive. To elaborate, suppose that we consider the recovery of s stripes, and there are n_j valid recovery solutions for recovering the lost chunk in the j-th stripe ($1 \leq j \leq s$). Then the enumeration approach would require $n_1 \times n_2 \times \cdots \times n_s$ trials. Depending on the number of valid recovery solutions in each stripe, the enumeration approach can involve a significantly large number of trials.

To mitigate the computation complexity into smaller, we propose a greedy algorithm to search for a near-optimal multi-stripe recovery solution for balancing the amount of cross-rack repair traffic across racks. Having a greedy recovery algorithm enables us to identify recovery solutions *on the fly*, especially under a dynamic environment with constant changing network conditions (e.g., the changing available network bandwidth) [39], [40]. The main idea is to iteratively replace the currently selected multi-stripe recovery solution with another one that introduces a smaller load balancing rate.

Algorithm 2 shows the details of our greedy algorithm. Suppose that a node in A_f fails ($1 \leq f \leq r$). We first select a valid recovery solution to repair the lost chunk in each stripe, and construct an initial multi-stripe recovery solution $\mathbb{R}$ (steps 1-3). Here, for each stripe, we can follow Theorem 1 to choose the valid recovery solution whose intact racks have



(a) Initial recovery solution: the load balancing rate is $\frac{16}{9}$.



(b) Recovery solution after a replacement: the load balancing rate is $\frac{12}{9}$.

Fig. 6. Example of how to substitute a per-stripe recovery solution in Algorithm 2. The chunks with the same color and fill patterns denote the retrieved chunks for recovery of the same stripe. Compared with the initial multi-stripe recovery solution, the updated multi-stripe recovery solution has a lower load balancing rate, by substituting the per-stripe recovery solution for the third stripe.

the most chunks for the stripe. We then replace the per-stripe recovery solutions in $\mathbb{R}$ over a configurable number of iterations (denoted by e), so as to reduce the load balancing rate. Specifically, in each iteration, we locate the rack A_l ($1 \leq l \neq f \leq r$) with the highest $t_{l,f}$ (i.e., generating the most cross-rack recovery traffic) (steps 4-5). To find a more balanced recovery solution, we scan the remaining intact racks except A_l and select one of the intact racks A_i ($i \leq l$ and $1 \leq i \neq f \leq r$) that satisfies the following condition:

$$t_{l,f} - t_{i,f} \geq 2. \tag{8}$$

Once identifying A_l and A_i, the algorithm scans the current per-stripe recovery solutions in $\mathbb{R}$. If the per-stripe recovery solution R_j for the j-th stripe ($1 \leq j \leq s$) reads chunks from rack A_l, then we check if there exists another valid recovery solution R'_j that can read chunks in A_i, meaning that it can substitute the retrieval from A_l (step 8). If both R_j and R'_j exist, we can substitute R_j with R'_j (steps 9-11). With partial decoding (see Section IV-C), we ensure that we retrieve one less partially decoded chunk from A_l while one more from A_i. Thus, Equation (8) ensures that $t_{l,f} \geq t_{i,f}$ after the substitution, and that the rack with the maximum amount of cross-rack repair traffic generated by a rack is monotonically decreasing. After the substitution, the algorithm resumes another iteration of the for-loop in Step 4 (step 11). If there is no substitution in $\mathbb{R}$, the algorithm exits the for-loop (step 12). As Algorithm 2 proceeds, the load balancing rate λ of $\mathbb{R}$ iteratively decreases.

Figure 6 shows an example of how our load balancing scheme works. We consider a CFS that has the same architecture and data layout as in Figure 1. The CFS also employs the $(k = 8, m = 6)$ RS code for fault tolerance. For brevity, we only illustrate the chunks retrieved for recovery. Suppose that the first node fails, Figure 6(a) first gives an initial multi-stripe recovery solution that recovers the lost chunks of four stripes. With partial decoding, the amount of cross-rack repair traffic can be represented by the number of partially decoded chunks transmitted from each intact rack. For example, A_2 sends four partially decoded chunks (i.e., $t_{2,1} = 4$) to recover the four lost chunks. Thus, the load balancing rate of the initial recovery solution is $\lambda = \frac{t_{2,1}}{(t_{2,1}+t_{3,1}+t_{4,1}+t_{5,1})/4} = \frac{16}{9}$. Obviously, in Figure 6(a), A_2 (i.e., A_l in Algorithm 2) is the rack with the most cross-rack traffic $t_{2,1} = 4$ (i.e., $t_{l,f}$). To find a more balanced solution, Algorithm 2 locates A_3 that satisfies the condition $t_{2,1} - t_{3,1} = 3 \geq 2$. The algorithm selects the per-stripe recovery solution for the third stripe, such that it retrieves a partially decoded chunk from A_3 instead of A_2. Figure 6(b) shows the new multi-stripe recovery solution. We can see that after the substitution, the load balancing rate of the updated recovery solution is $\lambda = \frac{t_{2,1}}{(t_{2,1}+t_{3,1}+t_{4,1}+t_{5,1})/4} = \frac{12}{9}$, which is smaller than that in Figure 6(a).

Complexity analysis: We now analyze the complexity of Algorithm 2. In each iteration, the algorithm finds the intact rack with the most cross-rack repair traffic, and search for another intact rack and per-stripe recovery solution for substitution (steps 6-11). The whole iteration needs no more than $r \times s$ trials. Since the algorithm repeats e iterations, its overall complexity is $O(e \times r \times s)$, which is in polynomial time.

V. Performance Evaluation

We conduct extensive testbed experiments to evaluate the performance of CAR. We would like to answer the following four questions:

1) How much cross-rack traffic and recovery time can be reduced by CAR?
2) How do the iteration steps affect the load balancing rate?
3) Will CAR sustain its effect when deployed over different CFS configurations?
4) Will CAR increase the computation time for recovery?

Evaluation environment: We conduct our evaluation on three CFS settings with different architectures and RS code parameters. Table II shows the configurations of the CFS settings for our evaluation, including the selected RS codes and the number of nodes in each rack. For example, CFS1 is deployed over three racks with 10 nodes and it selects the $(k = 4, m = 3)$ RS code. Note that practical storage systems often prefer a small number of nodes of a stripe (i.e., $k + m$) to avoid generating huge repair traffic. Thus, we configure the parameter $k + m$ to range from 7 to 14, such that this range covers typical system configurations of

TABLE II
CONFIGURATIONS OF THREE CFS SETTINGS.

CFSes	A_1	A_2	A_3	A_4	A_5	RS code
CFS1	4	3	3			$k = 4, m = 3$
CFS2	4	3	3	3		$k = 6, m = 3$
CFS3	6	4	5	3	2	$k = 10, m = 4$

TABLE III
CONFIGURATIONS OF NODES IN EACH RACK.

Servers	CPU	Memory	OS	Disk
Nodes in A_1	AMD Opteron(tm) 800MHz 2378 Quad-Core processors	16GB	Fedora 11	1TB
Nodes in A_2	an Intel Xeon X5472 3.00GHz Quad-Core CPU	8GB	SUSE Linux Enterprise Server 11	4TB
Nodes in A_3	an Intel Xeon E5506 2.13GHz Quad-Core CPU	8GB	Fedora 10	1TB
Nodes in A_4	an Intel Xeon E5420 2.50GHz Quad-Core CPU	4GB	Fedora 10	300GB
Nodes in A_5	an Intel Xeon X5472 3GHz Quad-Core CPU	8GB	Ubuntu 10.04.3 LTS	4TB

existing storage systems [1], [3]. For example, CFS2 selects the ($k = 6, m = 3$) RS code, which is consistent with Google Colossus FS [1]; CFS3 chooses the ($k = 10, m = 4$) RS code, which is the same as in Facebook's HDFS-RAID [3].

Table III also lists the hardware configurations of the nodes in different racks. We configure the nodes in the same rack to have the same hardware configurations. The racks are connected by the TP-LINK TL-SG1016D 16-Port Gigabit Ethernet switches. We also implement RS codes with the open-source erasure coding library Jerasure 1.2 [23].

Methodology: We construct 100 stripes and randomly distribute the data and parity chunks of each stripe across all nodes in each CFS, while ensuring single-rack fault tolerance (see Section IV-B). To evaluate the recovery performance, we randomly select a node to erase its stored chunks. We use the same node as the replacement node, and trigger the recovery operation. We apply CAR to find the recovery solution and recover the lost chunk of each stripe. For comparisons, we also consider a baseline approach called *random recovery* (RR), which finds the recovery solution by randomly choosing k surviving chunks of a stripe and sending them to the replacement node for recovery. To start recovery, the replacement node first contacts k surviving nodes for each stripe to simultaneously launch the transmissions of the chunks. For CAR, the replacement node also selects a node in each rack to perform partial decoding, such that the surviving nodes first send their chunks to the selected node in each rack for partial decoding, and then the selected node in each rack sends the aggregated chunk to the replacement node. On the other hand, for RR, the k surviving nodes directly send the chunks to the replacement node. Each of our results is averaged over 50 runs.

A. Cross-Rack Repair Traffic

We first evaluate the amounts of cross-rack repair traffic due to CAR and RR when recovering a single lost chunk. We conduct the evaluation in the three CFS settings. Figure 7 shows the results versus the chunk size. We make the following observations.

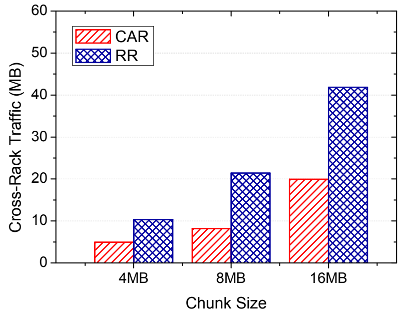
In all cases, CAR significantly reduces the amount of cross-rack repair traffic when compared to RR. For example, when the chunk size is 4MB, CAR can reduce 52.4% of cross-rack repair traffic in CFS1 (see Figure 7(a)). The reason is that CAR not only finds the recovery solution that involves the minimum number of racks, but also performs partial decoding in each rack before cross-rack data transmissions. Both techniques guarantee the minimum amount of cross-rack data transmissions when reconstructing the lost chunk in each stripe. As a comparison, RR simply retrieves the chunks from other surviving nodes to the replacement node, thereby triggering a considerable amount of cross-rack repair traffic.

In addition, the performance gain of CAR is influenced by the parameter k used in RS codes. In general, when the number of racks is fixed, CAR can reduce more cross-rack data transmissions when k increases. The reason is that in RR, the number of retrieved chunks increases when k becomes larger. On the other hand, CAR ensures that each rack only needs to send one chunk across racks under partial decoding. For example, when the chunk size is 16MB, the saving of cross-rack repair traffic due to CAR increases to 66.9% in CFS3 (see Figure 7(c)).

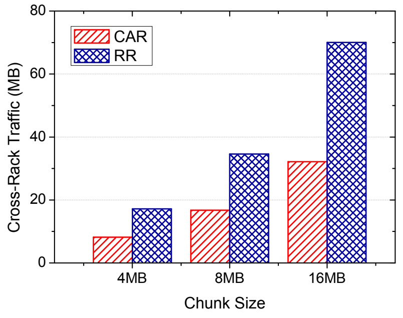
B. Load Balancing

In this evaluation, we measure the capability of CAR to balance the amount of cross-rack repair traffic across multiple racks. We configure the number of iterations (i.e., e) to be 50 and the number of stripes (i.e., s) to be 100 in Algorithm 2. In each CFS setting, we measure the load balancing rate (i.e., λ) of CAR after each number of iterations.

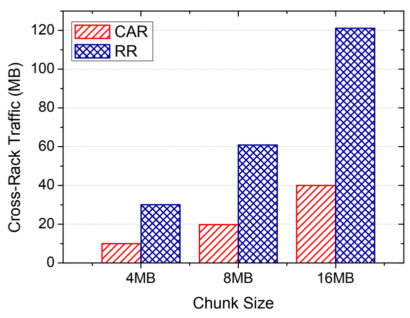
Figure 8 presents the average results and the standard deviations for CAR with and without performing load balancing (the latter means that we do not execute Algorithm 2). In all cases, CAR can effectively balance the amount of cross-rack repair traffic. For example, in CFS1 (see Figure 8(a)), if we do not perform load balancing, the load balancing rate is 1.22 even though CAR retrieves chunks from the minimum number of racks and performs partial decoding. With load balancing enabled, the load balancing rate of the optimized solution can reduce to 1.02. In addition, as we increase the number of iterations, the load balancing rate first decreases significantly and then becomes stable, mainly because the reduction in the load balancing rate in each iteration becomes smaller when the resulting solution is closer to the minimum.



(a) CFS1 (10nodes, $k = 4, m = 3$)

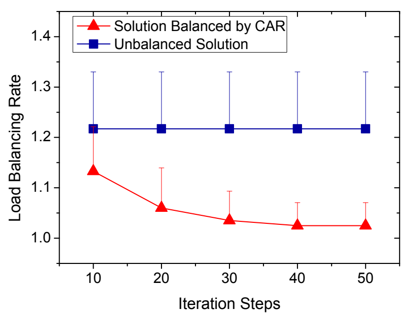


(b) CFS2 (13nodes, $k = 6, m = 3$)

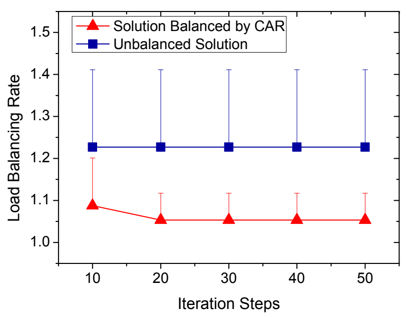


(c) CFS3 (20nodes, $k = 10, m = 4$)

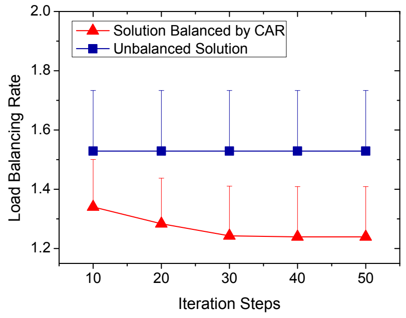
Fig. 7. Comparisons of the amounts of cross-rack traffic between CAR and RR.



(a) CFS1 (10nodes, $k = 4, m = 3$)

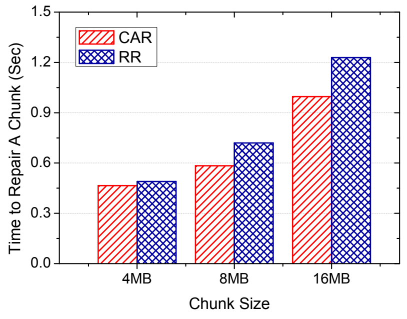


(b) CFS2 (13nodes, $k = 6, m = 3$)

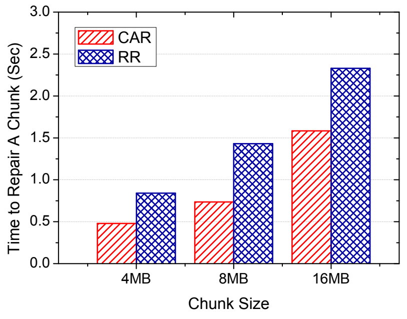


(c) CFS3 (20nodes, $k = 10, m = 4$)

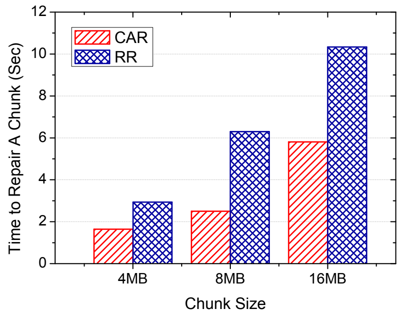
Fig. 8. Load balancing rate (and the standard deviation) versus the number of iteration steps in CAR. For brevity, we only show the standard deviations in the positive direction.



(a) CFS1 (10nodes, $k = 4, m = 3$)



(b) CFS2 (13nodes, $k = 6, m = 3$)



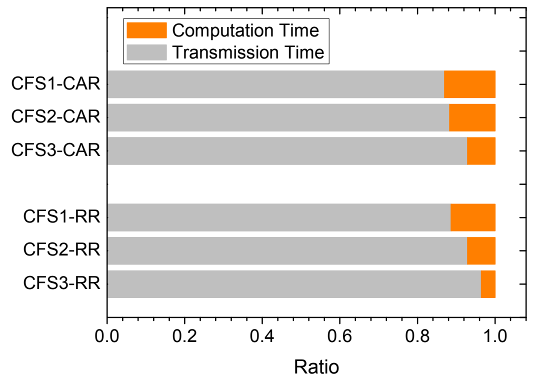
(c) CFS3 (20nodes, $k = 10, m = 4$)

Fig. 9. Comparisons of recovery times between CAR and RR.

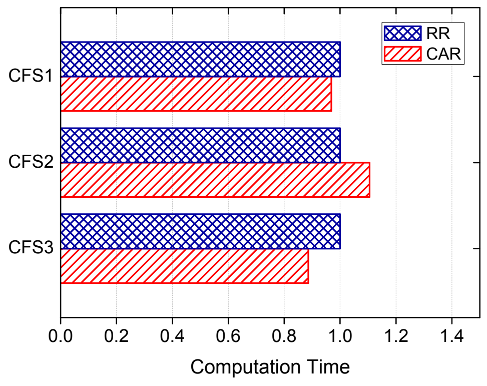
C. *Recovery Time*

We now compare CAR and RR in terms of the recovery time per lost chunk in different CFS settings. We measure the overall duration starting from the time when all surviving nodes send the chunks until the time when all lost chunks are completely reconstructed. We divide the overall duration by the number of lost chunks being reconstructed to obtain the recovery time per lost chunk.

Figure 9 shows the recovery time per lost chunk versus the chunk size. It indicates that CAR greatly reduces the recovery time when compared to RR. For example, when the chunk size is 8MB, to recover a lost chunk in CFS2, CAR reduces 53.8% of recovery time (see Figure 9(b)). The reasons are three-fold. First, CAR reduces the amount of cross-rack repair traffic. Second, CAR balances the amount of cross-rack repair traffic across multiple racks, while RR randomly selects k surviving chunks to recover a lost chunk and hence leads to an uneven distribution of cross-rack repair traffic in general. Third, CAR offloads the recovery process to a node in each rack due to partial decoding, while RR requires the replacement node to



(a) Ratios of transmission time and computation time



(b) Computation time (normalized with respect to that of RR)

Fig. 10. Evaluation of transmission time and computation time for recovering a lost chunk.

perform the whole recovery process for all lost chunks. Finally, we observe that a larger k in RS codes will increase the recovery time, mainly because it introduces more repair traffic for recovering each lost chunk.

D. Computation Time and Transmission Time

We further provide a breakdown on the recovery time, in terms of the transmission time and the computation time to recover a lost chunk. The transmission time records the duration of data transmissions over the CFS, while the computation time records the duration to perform required decoding operations over finite fields for reconstructing the lost chunk at the replacement node. We fix the chunk size as 8MB.

Figure 10 presents the results. Figure 10(a) shows that the transmission time dominates the overall recovery time, justifying the need of reducing the transmission overhead in CAR. Also, the ratio of computation time in both RR and CAR decreases when the parameter k in RS codes increases. For example, for CAR in CFS1 (where $k = 4$), the computation time occupies 11.3% of recovery time, while in CFS3, the ratio decreases to 7.1% (where $k = 10$).

Figure 10(b) shows that the computation time of CAR normalized over that of RR. In general, the computation times of both CAR and RR are similar (e.g., with up to around 10% of difference). Note that CAR does not change the decoding operations in RS codes, but instead only breaks down a decoding operation into multiple intra-rack partial decoding operations.

VI. Conclusions

Erasure coding is increasingly used to maintain data availability with low redundancy overhead in practical storage systems. This paper reconsiders the single failure recovery problem in a clustered file system (CFS), in which cross-rack bandwidth is often over-subscribed and considered to be a scarce resource. We propose CAR, a *cross-rack-aware recovery* algorithm that specifically addresses the single failure recovery problem in a CFS. CAR includes three key techniques. First, CAR examines the data layout in a CFS and determines the recovery solution that accesses the minimum number of racks for each stripe. Second, CAR performs partial decoding by aggregating the requested chunks in the same rack before cross-rack data transmissions. Third, CAR uses a greedy algorithm to find the recovery solution that balances the amount of cross-rack repair traffic across racks. Results from our testbed experiments show that CAR can reduce both cross-rack data transmissions and the overall recovery time.

Acknowledgments

The authors would like to thank the anonymous reviewers for their helpful comments and suggestions. This work is supported by the National Natural Science Foundation of China (Grant Nos. 61232003, 61433008), the Beijing Municipal Science and Technology Commission of China (Grant No. D151100000815003), the State Key Laboratory of High-end Server and Storage Technology (Grant No. 2014HSSA02), the University Grants Committee of Hong Kong (Grant No. AoE/E-02/08), Cisco University Research Program Fund (CG#593756) from Silicon Valley Community Foundation, and Research Committee of CUHK.

References

[1] Colossus, successor to google file system. http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf.

[2] J. Bloemer, M. Kalfane, R. Karp, M. Karpinski, M. Luby, and D. Zuckerman. An xor-based erasure-resilient coding scheme. Technical report, 1995.

[3] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling. Hdfs raid. In *Hadoop User Group Meeting*, 2010.

[4] B. Calder, J. Wang, A. Ogus, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proc. of ACM SOSP*, 2011.

[5] J. C. Chan, Q. Ding, P. P. Lee, and H. H. Chan. Parity logging with reserved space: towards efficient updates and recovery in erasure-coded clustered storage. In *Proc. of USENIX FAST*, 2014.

[6] M. Chowdhury, S. Kandula, and I. Stoica. Leveraging endpoint flexibility in data-intensive clusters. In *Proc. of ACM SIGCOMM*, 2013.

[7] P. Corbett, B. English, A. Goel, T. Grcanac, S. Kleiman, J. Leong, and S. Sankar. Row-diagonal parity for double disk failure correction. In *Proc. of USENIX FAST*, 2004.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *USENIX OSDI*, 2004.

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

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

[11] Y. Fu, J. Shu, and X. Luo. A stack-based single disk failure recovery scheme for erasure coded storage systems. In *Proc. of IEEE SRDS*, 2014.

[12] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of ACM SOSP*, 2003.

[13] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in windows azure storage. In *Proc. of USENIX ATC*, 2012.

[14] C. Huang and L. Xu. Star: An efficient coding scheme for correcting triple storage node failures. *Computers, IEEE Transactions on*, 57(7):889–901, 2008.

[15] C. Jin, H. Jiang, D. Feng, and L. Tian. P-code: A new raid-6 code with optimal properties. In *Proc. of ACM ICS*, 2009.

[16] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads. In *Proc. of USENIX FAST*, 2012.

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

[18] R. Li, Y. Hu, and P. P. Lee. Enabling efficient and reliable transition from replication to erasure coding for clustered file systems. In *Proc. of IEEE/IFIP DSN*, 2015.

[19] R. Li, P. P. Lee, and Y. Hu. Degraded-first scheduling for mapreduce in erasure-coded storage clusters. In *IEEE/IFIP DSN*, 2014.

[20] X. Luo and J. Shu. Load-balanced recovery schemes for single-disk failure in storage systems with any erasure code. In *Proc. of IEEE ICPP*, 2013.

[21] S. Muralidhar, W. Lloyd, S. Roy, et al. F4: Facebooks warm blob storage system. In *Proc. of USENIX OSDI*, 2014.

[22] J. S. Plank, M. Blaum, and J. L. Hafner. Sd codes: erasure codes designed for how storage systems really fail. In *Proc. of USENIX FAST*, 2013.

[23] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2. *University of Tennessee, Tech. Rep. CS-08-627*, 23, 2008.

[24] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran. Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth. In *Proc. of USENIX FAST*, 2015.

[25] K. 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. of ACM SIGCOMM*, 2014.

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

[27] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. Dimakis, R. Vadali, S. Chen, and D. Borthakur. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, 2013.

[28] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *Proc. of USENIX FAST*, 2002.

[29] B. Schroeder and G. Gibson. Disk failures in the real world: What does an mttf of 1, 000, 000 hours mean to you? In *Proc. of USENIX FAST*, 2007.

[30] Z. Shen and J. Shu. Hv code: An all-around mds code to improve efficiency and reliability of raid-6 systems. In *Proc. of IEEE/IFIP DSN*, 2014.

[31] Z. Shen, J. Shu, and Y. Fu. Seek-efficient i/o optimization in single failure recovery for xor-coded storage systems. In *Proc. of IEEE SRDS*, 2015.

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

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

[34] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, and C. Xie. Hdp code: A horizontal-diagonal parity code to optimize i/o load balancing in raid-6. In *Proc. of IEEE/IFIP DSN*, 2011.

[35] M. Xia, M. Saxena, M. Blaum, and D. A. Pease. A tale of two erasure codes in hdfs. In *Proc. of USENIX FAST*, 2015.

[36] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in rdp code storage systems. In *Proc. of ACM SIGMETRICS*, 2010.

[37] L. Xu and J. Bruck. X-code: Mds array codes with optimal encoding. *IEEE Transactions on Information Theory*, 45(1):272–276, 1999.

[38] S. Xu, R. Li, P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. Lui. Single disk failure recovery for x-code-based parallel storage systems. *IEEE Trans. on Computers*, 63(4):995–1007, 2014.

[39] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu. On the speedup of single-disk failure recovery in xor-coded storage systems: Theory and practice. In *Proc. of IEEE MSST*, 2012.

[40] Y. Zhu, P. P. Lee, L. Xiang, Y. Xu, and L. Gao. A cost-based heterogeneous recovery scheme for distributed storage systems with raid-6 codes. In *Proc. of IEEE/IFIP DSN*, 2012.