This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

Cross-Rack-Aware Single Failure Recovery for Clustered File Systems

Zhirong Shen, Patrick P. C. Lee, Jiwu Shu, and Wenzhong Guo

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 clustered 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 propose CAR, a *cross-rack-aware recovery* algorithm that is designed to improve the performance of single failure recovery of a CFS that employs Reed-Solomon codes for general fault tolerance. 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.

1 Introduction

To process an ever-increasing amount of data, distributed computing applications often build on a *clustered file system (CFS)*, which provides a unified and scalable storage platform. A CFS is constructed over a number of physically independent storage servers, which we refer to as *nodes* in this paper. Examples of CFSes include Google File System [13], Hadoop Distributed File System [37], and Windows Azure Storage [5].

Failures are commonplace in large-scale CFSes [8], [11], [13], [32], [33]. In particular, most failures found in CFSes are *single node failures* (or single failures in short), which can occupy over 90% of all failure events in real deployment [11]. To maintain data availability, 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 [38], and hence is increasingly used in today's CFSes for improved storage efficiency. The mainstream approach of erasure coding takes original pieces of information of the same size (termed *data chunks*) as inputs and produces redundant pieces of information that are also of the same size (termed *parity chunks*), such that if a data or parity chunk is lost, we can still retrieve other available data and parity chunks to reconstruct the lost chunk. The collection of data and parity chunks that are connected by the erasure coding forms a *stripe*. A CFS stores multiple stripes of information, each of which is independently encoded.

Given the prevalence of single failures, there have been a spate of solutions (e.g., [12], [18], [28], [29], [35], [40], [42], [43]) 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 (see Section 2.4 for details). First, typical CFS architectures organize nodes in multiple racks, yet existing studies on single failure recovery neglect the *bandwidth diversity* property between intra-rack and cross-rack connections in a CFS architecture. Second, many single failure recovery solutions (e.g., [12], [40], [42], [43]) focus on specific code constructions, but cannot be directly applied to today's CFSes (e.g., [1], [11], [24]) that employ Reed-Solomon (RS) codes [30] for general fault tolerance. Third, existing single failure recovery solutions do not consider the load balancing issue during the recovery operation.

To address the above limitations, we reconsider the single failure recovery problem in a CFS setting. First, we should specifically minimize the *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 performance of single failure recovery 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 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 performance of single failure recovery is not bottlenecked by a single rack.

To this end, we propose *cross-rack-aware recovery* (CAR),

- *An earlier version of this paper appeared at the 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'16) [36]. In this journal version, we include additional analysis on rack-level fault tolerance and more evaluation results for CAR.*
- *Zhirong Shen and Patrick P. C. Lee are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong (Email: zhirong.shen2601@gmail.com, pclee@cse.cuhk.edu.hk).*
- *Jiwu Shu is with the Department of Computer Science and Technology, Tsinghua University, Beijing, China (Emails: shujw@tsinghua.edu.cn).*
- *Wenzhong Guo is with the College of Mathematics and Computer Science, Fujian Provincial Key Laboratory of Network Computing and Intelligent Information Processing, Fuzhou University, China (Emails: guowenzhong@fzu.edu.cn).*
- *Corresponding author: Jiwu Shu (shujw@tsinghua.edu.cn).*

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

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, so as to further reduce the amount of cross-rack repair traffic. Finally, 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 cross-rack repair traffic across racks. We also discuss how CAR should be deployed to achieve general rack-level fault tolerance.
- 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 47.9% 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 multiple racks.

The rest of this paper proceeds as follows. Section 2 presents the background details of erasure coding and reviews related work on single failure recovery. Section 3 formulates and motivates the problem in the CFS setting. Section 4 presents the design of CAR. Section 5 presents our evaluation results on CAR based on testbed experiments. Finally, Section 6 concludes the paper.

2 Background and Related Work

2.1 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 (ToR) switch*, while all racks are connected by a *network core* that comprises layers of aggregation switches. 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 [13], Hadoop Distributed File System [37], and Windows Azure Storage [5], realize the CFS architecture. Such a CFS architecture is also considered in the literature (e.g., [6], [20]).

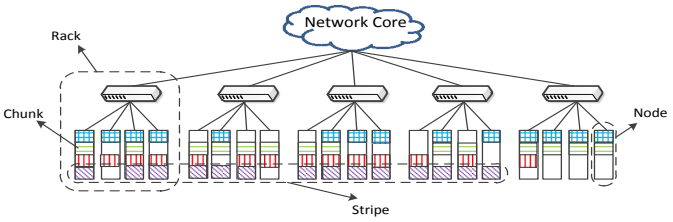


Fig. 1. Illustration of a CFS architecture, composed of five racks with four nodes each. The CFS contains four stripes of 14 chunks encoded by a $(k = 8, m = 6)$ code, in which the chunks with the same color and fill pattern belong to the same stripe. Note that the number of chunks in each node may be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

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 [20], such that there are at least k chunks for data reconstruction in other surviving racks in the presence of rack failures.

For an erasure-coded CFS that stores a large amount of data, it contains multiple stripes of data/parity chunks that are independently encoded. 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.

2.2 Erasure Code Constructions

There have been various proposals on erasure code construction in the literature. Practical erasure codes often realize encoding/decoding operations based on the arithmetic over the Galois field [27]. Reed Solomon (RS) codes [30] 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 (i.e., $k = 6$) 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 [4]. 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 [41], STAR Code [17], and HV Code [34]. 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 $(k = p - 1, m = 2)$,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

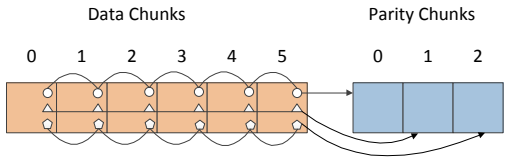


Fig. 2. Encoding of ($k = 6, m = 3$) RS codes for a stripe, in which there are six data chunks and three parity chunks. If one of the data or parity chunks fails, any six surviving chunks within the stripe can be retrieved for reconstruction.

X-Code [41] requires ($k = p - 2, m = 2$), and STAR Code [17] 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 [11], [16]. 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 [10] 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 [10] are MDS, and they minimize the amount of repair traffic subject to the minimum storage redundancy. Rashmi et al. [28] propose a new MSR code construction that also minimizes the amount of I/Os. Huang et al. [16] and Sathiamoorthy et al. [31] 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 [25] and STAIR Codes [19]. They are non-MDS, and perform encoding/decoding operations over the Galois field. They are mainly designed for local disk arrays.

2.3 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 2.2), previous studies (e.g., [12], [18], [22], [40], [43]) 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. [40] and Xu et al. [42] 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. [18] propose to enumerate all possible single failure recovery solutions and select the one that minimizes the amount of I/Os. Luo et al. [22] and Fu et al. [12] extend the enumeration approach of Khan et al. [18] to 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. [43] and Shen et al. [35] propose a greedy algorithm to search for the single failure recovery solution with the near-minimum amount of I/Os, while still supporting general XOR-based erasure codes.

Some studies also address the performance issue when deploying erasure codes in a CFS. For example, Li et al. [20] 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. Xia et al. [39] present a new approach to switch between two erasure codes to balance the storage overhead and recovery performance.

There are recent studies that are closely related to ours. Mitra et al. [23] propose *Partial Parallel Repair (PPR)*, which decomposes a recovery operation into many small partial operations and schedules those partial operations in parallel, so as to achieve faster data recovery. The use of partial recovery operations is similar to our partial decoding approach (see Section 4.2), but PPR does not address how to minimize the cross-rack repair traffic. Li et al. [21] propose to minimize the single failure recovery time by dividing chunks into small slices and pipelining the repair of these slices. Hu et al. [14], [15] design Double Regenerating Codes (DRC) to use two stages to minimize the cross-rack repair traffic, while achieving the minimum storage efficiency. Their objective is similar to ours: Hu et al. [14], [15] focus on deriving a new erasure code construction that matches the optimal point of regenerating codes [10], while our work specifically focuses on RS codes, which have been widely deployed in current CFSes, and accordingly proposes inherently different recovery techniques.

2.4 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 in a CFS setting. We have provided an overview of the open issues in Section 1, and following discussion provides more detailed explanations.

2.4.1 Lack of Considerations on Cross-rack Repair Traffic

Existing single failure recovery optimizations [12], [18], [22], [35], [40], [42], [44], 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 intra-rack bandwidth is considered to be sufficient, while cross-rack bandwidth is often *over-subscribed*. The typical values of the over-subscription ratio in a data center network are in the range from 5 to 20 [2]. Thus, cross-rack bandwidth is often considered to be a scarce resource [6], [9], [20]. A recovery solution that triggers a large amount

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

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.

2.4.2 Ineffectiveness for RS Codes

Most existing studies [12], [18], [22], [35], [40], [42], [44] on single failure recovery mainly focus on XOR-based erasure codes. While XOR-based erasure codes achieve high encoding/decoding performance by only using XOR operations, they are not the common choice in a CFS due to their 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], [11], [24]) usually employ Reed-Solomon (RS) codes [30] for general fault tolerance. RS codes perform encoding/decoding operations over finite fields [27], and have inherently different constructions from XOR-based erasure codes. In general, RS codes reconstruct a lost chunk by retrieving any k surviving chunks within the same stripe. This strategy implies 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.

2.4.3 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 2.1). 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 [12], [35], [42]. However, the load balancing schemes [12], [35], [42] 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.

3 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 1 summarizes the major notation used in this paper.

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 $s \geq 1$ stripes. We denote the rack that contains the failed node as A_f ($1 \leq f \leq r$); 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 $t_{i,f}$ ($1 \leq i \neq f \leq r$). 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.

TABLE 1
Major notations 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 ($1 \leq i \leq r$) rack
A_f	the rack where the failed node resides ($1 \leq f \leq r$)
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
u	tolerable number of failed racks ($1 \leq u \leq m$)

$$\lambda = \frac{\max\{t_{i,f} | 1 \leq i \neq f \leq r\}}{\sum_{1 \leq i \neq f \leq r} \frac{t_{i,f}}{r-1}}.$$

Obviously, if there exists cross-rack repair traffic, then $\lambda \geq 1$. 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

$$\sum_{1 \leq i \neq f \leq r} t_{i,f} \text{ is minimized.}$$

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.

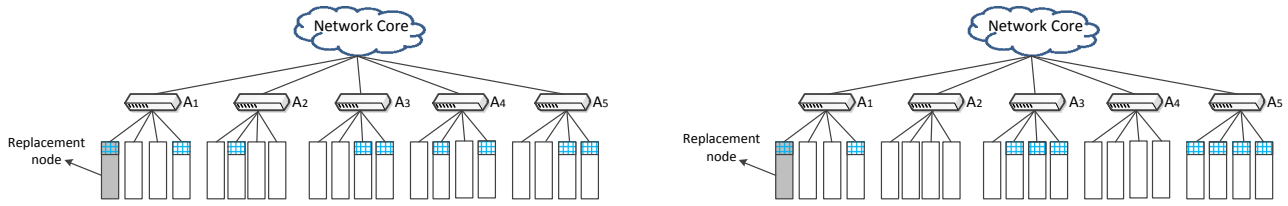
4 Cross-rack-aware Recovery

We thus present CAR, a *cross-rack-aware recovery* algorithm. 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 identify a recovery solution that retrieves chunks from the minimum number of racks. To repair the lost chunk, instead

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing



(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.

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. Thus, 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_3 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.

4.1 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 [20]. Here, we assume that we provide single-rack fault tolerance, and we address multi-rack fault tolerance in Section 4.4. 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:

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

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:*

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

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

Proof: 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 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. □

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 $(1 \leq j \leq s)$ by accessing d_j intact racks only. A valid solution of the j-th stripe $(1 \leq j \leq$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

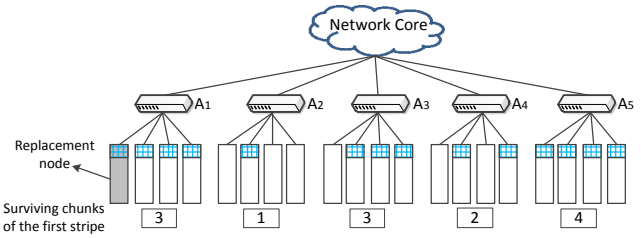


Fig. 4. Example of determining the minimum number of intact racks to be accessed when recovering the lost chunk in the first stripe. Suppose that the CFS employs the $(k = 8, m = 6)$ RS code, and that the first node in A_1 fails. The replacement node can retrieve chunks from the intact racks A_3 and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

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.

4.2 Intra-rack Chunk Aggregation

After finding the minimum number of intact racks to be accessed for recovery, we perform intra-rack chunk aggregation on the retrieved chunks in the same rack. We call the aggregation operation *partial decoding*, as 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 2.1), 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 $\mathcal{G} = \left(\mathbf{g}_1 \cdots \mathbf{g}_{k+m}\right)^T$ 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, $\mathbf{g}_i$ $(1 \leq i \leq k + m)$ is a row vector and its size is $1 \times k$. To make the original data kept in uncoded form, $\left(\mathbf{g}_1 \cdots \mathbf{g}_k\right)^T$ 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 $(1 \leq i \leq k+m)$, 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, $\mathbf{y} = \left(y_1 \cdots y_k\right)$ is a $1 \times k$ vector. Then we can derive the following equation based on Equation (5).

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

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:

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

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 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 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. Then 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.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

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.

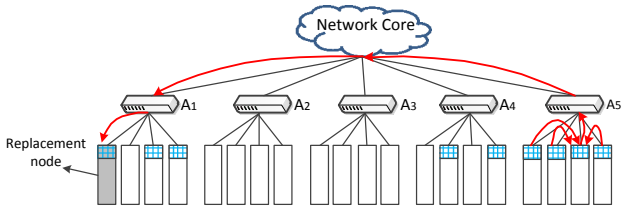


Fig. 5. Example of reconstructing the lost chunk in the first stripe via partial decoding. For example, four chunks in rack A_5 are selected for reconstruction. One node in A_5 performs partial decoding on the four selected chunks and sends the partially decoded chunk to the replacement node.

4.3 Load Balancing

As stated in Section 4.1, 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 3). 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) [43], [44]. 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 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 balance rate. The selection of e depends on both the system's computational capacity and the expected load balancing rate. In general, there are two ways to determine the value of e: (i) the system can choose the maximum value of e that is allowed, subject to the computational capacity constraint; or (ii) the system defines the expected gap of load balancing rates obtained in any two adjacent iterations, such that the algorithm terminates once the reduction of the load balancing rate is lower than the given gap. 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 repair 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 ($i \neq l$ 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 4.2), 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.

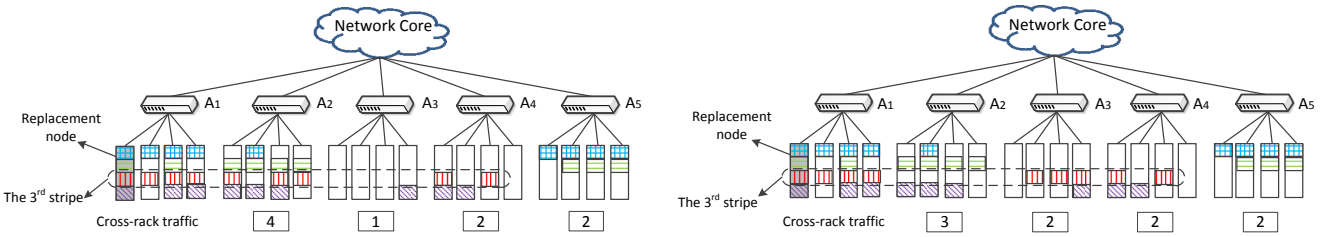
Figure 6 shows an example of how our load balancing scheme works. We consider a CFS that has the same architecture and data layout as in Figure 1. The CFS also

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 the 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 $\mathbb{R}$

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing



(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.

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 transmits 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.

4.4 Multi-rack Fault Tolerance

We thus far assume that CAR achieves only single-rack fault tolerance, by placing no more than m chunks in each rack (see Section 4.1). We now generalize the rack-level fault tolerance problem and examine how to arrange the chunk placement so that CAR can tolerate multiple rack failures.

Our insight is that using intra-rack chunk aggregation (see Section 4.2), we can reduce more cross-rack repair traffic by placing more chunks of each stripe in a rack. However, to provide rack-level fault tolerance, we must spread the chunks of each stripe across multiple racks. Our goal is to formalize the chunk placement requirement so as to distribute the chunks of each stripe as "compact" as possible (i.e., spanning the least number of racks), while satisfying the required rack-level fault tolerance.

We now define the notation. For a (k, m) RS code, we disperse the $k+m$ chunks of each stripe over r racks, where $1 \leq r \leq k+m$. Suppose that our goal is to tolerate u rack failures, where $1 \leq u \leq \min\{r, m\}$. Theorem 2 states how we place the chunks of a stripe over the minimum number of racks.

Theorem 2. *For a (k, m) RS code, we can tolerate the failures of any u out of r racks if and only if the number of racks spanned by a stripe satisfies the following condition:*

$$r \geq u + \lceil \frac{k}{\lfloor \frac{m}{u} \rfloor} \rceil. \tag{9}$$

Proof: We first prove the "Only if" part. Without loss of generality, we rank the numbers of chunks of the j-th stripe in the r racks in descending order denoted by $\{c_{j_1}, c_{j_2}, \cdots, c_{j_r}\}$, where $c_{j_1} \geq c_{j_2} \geq \cdots \geq c_{j_r}$. To tolerate any u rack failures, we require that

$$c_{j_1} + c_{j_2} + \cdots + c_{j_u} \leq m. \tag{10}$$

Equation (10) states that we must store no more than m chunks in any u racks for fault tolerance. From Equation (10), we can also show the following:

$$c_{j_u} \leq \frac{c_{j_1} + c_{j_2} + \cdots + c_{j_u}}{u} \leq \lfloor \frac{m}{u} \rfloor. \tag{11}$$

Suppose that the u racks that store the most chunks of the j-th stripe now all fail. The remaining $k + m - \sum_{i=1}^{u} c_{j_i}$ chunks are stored in the remaining $r - u$ surviving racks. Since each surviving rack stores no more than c_{j_u} chunks, the number of surviving racks $r - u$ must satisfy:

$$r - u \geq \lceil \frac{k + m - \sum_{i=1}^{u} c_{j_i}}{c_{j_u}} \rceil \geq \lceil \frac{k}{\lfloor \frac{m}{u} \rfloor} \rceil, \tag{12}$$

due to Equations (10) and (11). The "Only if" part holds.

We now prove the "If" part. The intuition is to find a chunk placement that satisfies Equation (9). We fix $r = r^*$, where $r^* = u + \lceil \frac{k}{\lfloor \frac{m}{u} \rfloor} \rceil$. To place $k + m$ chunks of the j-th stripe over r^* racks, we set $c_{j_2} = c_{j_3} = \cdots = c_{j_{r^*-1}} = \lfloor \frac{m}{u} \rfloor$, $c_{j_1} = m - (c_{j_2} + c_{j_3} + \cdots + c_{j_u})$, and $c_{j_{r^*}} = k - (c_{j_{u+1}} + \cdots + c_{j_{r^*-1}})$. We can easily verify that c_{j_1} is the largest, $c_{j_{r^*}}$ is the smallest and non-negative, and hence the list $c_{j_1}, c_{j_2}, \cdots, c_{j_{r^*}}$ are in descending order. Now, we can show that $c_{j_1} + c_{j_2} + \cdots + c_{j_u} \leq m$, and hence for every u out of r^* racks, there must be no more than m chunks. Thus,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

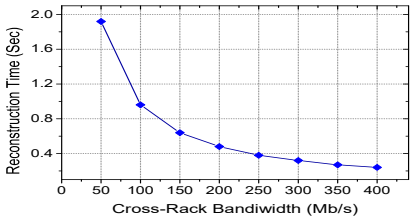


Fig. 7. The reconstruction time when the cross-rack bandwidth varies.

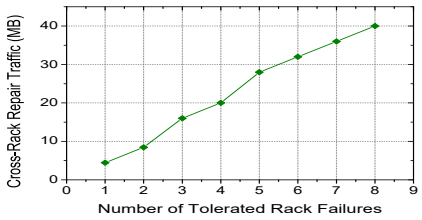


Fig. 8. The amount of cross-rack recovery traffic when the number of tolerated rack failures varies.

this chunk placement can tolerate any u rack failures. The "If" part holds. □

For example, suppose that we deploy a $(k = 7, m = 5)$ RS code and want to tolerate any $u = 2$ rack failures. Then we should choose $r^* = 2 + \lceil \frac{7}{\lfloor \frac{5}{2} \rfloor} \rceil = 6$ racks. We set $c_{j_2} = c_{j_3} = \cdots = c_{j_{r^*-1}} = \lfloor \frac{5}{2} \rfloor = 2$, $c_{j_1} = 5 - 2 = 3$, and $c_{j_{r^*}} = 7 - (2 + 2 + 2) = 1$, such that the distribution of chunks is $(3, 2, 2, 2, 2, 1)$. We can verify that this chunk placement can tolerate any $u = 2$ rack failures.

4.5 Analysis

We conduct analysis on the design implications of CAR under different settings.

4.5.1 Impact of Cross-Rack Bandwidth

We first analyze how the recovery time of CAR varies with the cross-rack bandwidth. Let C be the chunk size and B be the available cross-rack bandwidth. Here, we assume that the recovery time is mostly dominated by the cross-rack transfer time instead of by other factors, such as CPU encoding/decoding time and intra-rack transfer time. Suppose that CAR retrieves one chunk from each of other r' racks through intra-rack chunk aggregation (where $r' < r$). The recovery time of CAR is $\frac{r'C}{B}$. To illustrate the implication, suppose that all nodes are interconnected by a 1Gb/s Ethernet, and that the over-subscription ratio ranges from 2.5 to 20 (i.e., the cross-rack bandwidth ranges from 50Mb/s to 400Mb/s). Figure 7 plots the recovery time versus the cross-rack bandwidth. We can observe that the recovery time increases when the over-subscription ratio becomes larger (i.e., the cross-rack bandwidth is smaller).

4.5.2 Impact of Rack Fault Tolerance

We also analyze how CAR makes a design trade-off between the amount of cross-rack recovery traffic and the number of rack failures that can be tolerated (i.e., u). We select a $(k = 10, m = 8)$ erasure code and assume that the size of a chunk is 4MB. Given an expected number of tolerated rack failures, we first derive a chunk distribution based on the method in Section 4.4. We then measure the average amount of cross-rack repair traffic caused by CAR to recover each chunk. The results are shown in Figure 8, which indicates that the amount of cross-rack repair traffic increases as the number of tolerable rack failures increases.

Discussion: When under the same chunk distribution and failure rates, CAR needs less recovery time than the random recovery (i.e., randomly select k surviving chunks and directly send them to the replacement node, see Section 5 for details), and makes the system stay at the reliable state for longer time. Therefore, CAR can improve the system reliability when compared with traditional random recovery.

4.6 Extension of CAR

CAR mainly focuses on node recovery based on the network topology of CFSes. Nevertheless, the design principle of CAR can still provide a valuable reference for node recovery for general network topologies, provided that over-subscription exists. Specifically, if the recovery has to deliver the requested data to the replacement node over a bandwidth-limited connection, then the system can first perform partial decoding (as in CAR) and send the partial decoded result to the replacement node, so as to mitigate the congestion on the bandwidth-limited connection and reduce the overall recovery time.

Currently, CAR mainly focuses on RS code, and we accordingly design three techniques for CAR. In fact, the first two techniques (i.e., minimizing the number of accessed racks and intra-rack chunk aggregation) can also help to reduce the cross-rack data transfer if we use XOR-based erasure codes (see Section 2.2). We can first find the minimum number of accessed racks given the distribution of surviving chunks and the decoding rules of a specific XOR-based erasure code, followed by aggregating the data within each rack based on XOR operations. For load balancing, we may need a different greedy algorithm for XOR-based erasure codes; we pose it as future work.

5 Performance Evaluation

5.1 Implementation Overview

We implement a prototype of CAR in C on Linux. We implement RS codes, whose encoding and decoding operations are realized based on the open-source library Jerasure 1.2. [27]. We configure the size of a chunk in CAR ranging from 4MB to 16MB. In failure recovery, each selected chunk will be partitioned into many *sub-chunks* whose sizes are the order of kilobytes (e.g., 4KB), and the recovery operation is performed in a pipelined manner. In each rack, we select one node to perform partial decoding. Each other node (except the replacement node and the partial decoding node) will establish a socket connection and transfers a sub-chunk to the partial decoding node, which uses *aio* library [3] to perform asynchronous reads to collect the sub-chunks. After that, the partial decoding node computes the partial decoded sub-chunk based on the received sub-chunks and sends it to the replacement node via a socket connection. The replacement node will recover a sub-chunk after adding all the partial decoded sub-chunks. We repeat the procedure for all sub-chunks until the lost chunk is successfully repaired.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

TABLE 2
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 3
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

5.2 Evaluation Results

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

1) How much cross-rack traffic and the time of single failure recovery can be reduced by CAR?
2) Will CAR sustain its effectiveness when deployed over different CFS configurations, including the number of racks, the number of nodes per rack, and the erasure code parameters?
3) How do the iteration steps affect the load balancing rate?
4) Will CAR increase the computational overhead for recovery?

Evaluation environment[1]: We conduct our evaluation on three CFS settings with different architectures and RS code parameters. Table 2 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. In practical deployment, even a CFS contains a large number of nodes, the erasure coding parameters are often configured to make $k + m$ not to be too large so as to limit the encoding overhead and the amount of repair traffic, while maintaining fault tolerance [26]. For example, Google Colossus FS [1], [4] uses the $(k = 6, m = 3)$ RS Code and HDFS-RAID [1], [4] uses the $(k = 10, m = 4)$ RS Code. Thus, each stripe with $k + m$ chunks will only span a limited number of nodes in real-world storage systems, while multiple stripes are independently encoded and repaired under failures. We configure the stripe size $k + m$ in our evaluation to range from 7 to 14, such that this range covers typical system configurations of existing storage systems [1], [4]. Thus, we expect that our CFS configurations are sufficiently practical to reflect the repair performance in real-world deployment.

Table 3 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.

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 4.1). 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 multiple trials (generally 5 trails to 10 trails). We find that the standard deviation is small, so we do not plot the standard deviation in the figures.

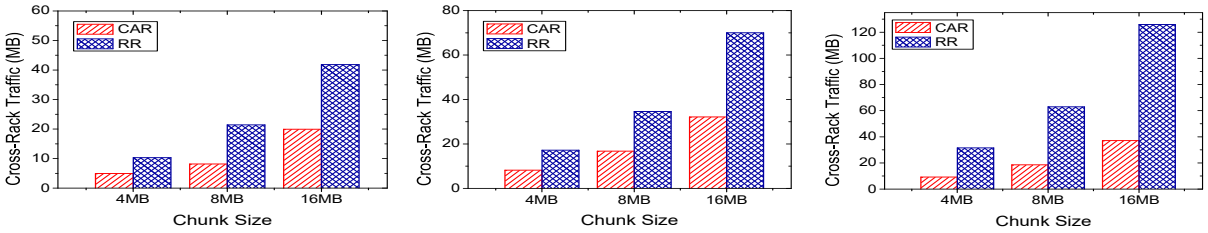
Experiment 1 (Recovery Performance in Different CFS Settings). We first evaluate the amounts of cross-rack repair traffic due to CAR and RR when recovering a single lost chunk. We conduct the evaluation in the three CFS settings shown in Table 2. Figure 9 shows the results of cross-rack traffic versus 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 9(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-

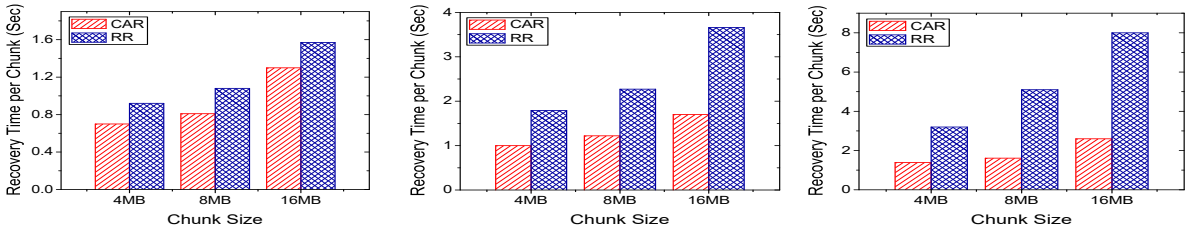
1. Note that we rerun all our experiments and hence our performance results is slightly different from those in our conference version [36], although the key conclusions remain the same.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing



(a) CFS1 (10 nodes, $k = 4, m = 3$) (b) CFS2 (13 nodes, $k = 6, m = 3$) (c) CFS3 (20 nodes, $k = 10, m = 4$)

Fig. 9. Experiment 1: Comparisons of the amounts of cross-rack traffic between CAR and RR over different CFSes.



(a) CFS1 (10 nodes, $k = 4, m = 3$) (b) CFS2 (13 nodes, $k = 6, m = 3$) (c) CFS3 (20 nodes, $k = 10, m = 4$)

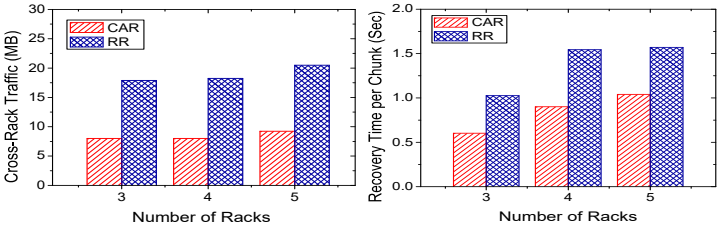
Fig. 10. Experiment 1: Comparisons of recovery times between CAR and RR over different CFSes.

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 9(c)).

We further 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 10 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 46.2% of recovery time (see Figure 10(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 perform the whole recovery process for all lost chunks.

Experiment 2 (Impact of Number of Racks). We next evaluate the performance of CAR when it is deployed over different number of racks. We consider a new CFS setting as follows. We select the $(k = 6, m = 3)$ RS code and perform the evaluation when the number of racks increases from three to five, in which we use $\{A_1, A_2, A_3\}$, $\{A_1, A_2, A_3, A_4\}$, and $\{A_1, A_2, A_3, A_4, A_5\}$ in Table 3, respectively. Each rack consists of three nodes, and the nodes in the same rack have the same hardware configurations and operating system (see Table 3). The chunk size is set as 4MB. In the evaluation, we erase the data on each node and



(a) Cross-rack repair traffic (b) Recovery time

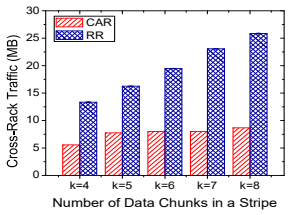
Fig. 11. Experiment 2: Impact of number of racks.

invoke failure recovery. We measure the average amounts of cross-rack repair traffic and recovery time to reconstruct a lost chunk. Figure 11 shows the evaluation results.

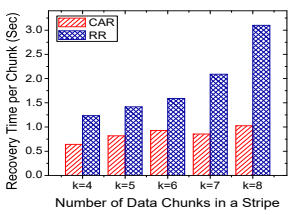
Figure 11(a) first presents the amount of cross-rack traffic to repair a lost chunk under different number of racks. We make three observations. First, there will be more cross-rack traffic in CAR when the number of racks increases. The reason is that the distribution of chunks will be more "sparse" (i.e., fewer chunks in a rack) if they are dispersed across more racks, which will limit the effectiveness of intra-rack chunk aggregation. Second, RR will also introduce more cross-rack repair traffic when the number of rack increases. As shown before, RR needs to directly read any k chunks to repair a lost chunk. Increasing the number of racks will place more chunks in the intact racks, and hence produce more cross-rack repair traffic. Third, CAR still keeps its effectiveness even when the number of racks varies. For example, compared with RR, CAR reduces about 55.3% of cross-rack repair traffic when the CFS has three racks. This saving will be 54.9% when the number of racks increases to five.

Figure 11(b) presents the average recovery time to repair a chunk versus the number of racks. We see that both CAR and RR incurs more recovery time when there are more racks. For example, CAR needs 0.60 seconds to repair a lost chunk when there are three racks, and the recovery time increases to 1.04 seconds when the number of racks is five. CAR still sees performance gain; for example, its recovery

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

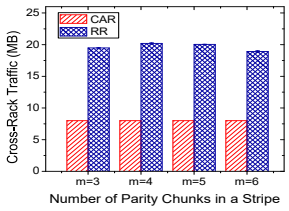


(a) Cross-rack repair traffic

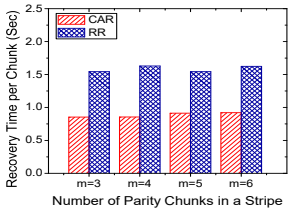


(b) Recovery time

Fig. 12. Experiment 3: Impact of number of data chunks.



(a) Cross-rack repair traffic



(b) Recovery time

Fig. 13. Experiment 3: Impact of number of parity chunks.

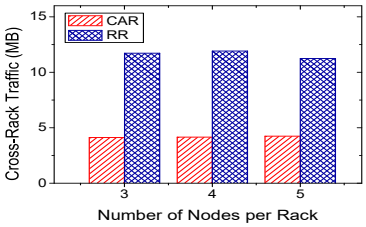
time is 33.7% less than that of RR when there are five racks.

Experiment 3 (Impact of Erasure Coding Configurations). We further investigate the impact of selected parameters in erasure coding schemes. We consider a new CFS setting as follows. We configure a CFS over four racks $\{A_1, A_2, A_3, A_4\}$, each of which includes three nodes with configurations shown in Table 3. We set the chunk size to be 4MB and select the $(k = 6, m = 3)$ RS code. We then evaluate the cross-rack traffic and the time to repair a lost chunk when k and m vary respectively. The evaluation results are respectively shown in Figures 12 and 13.

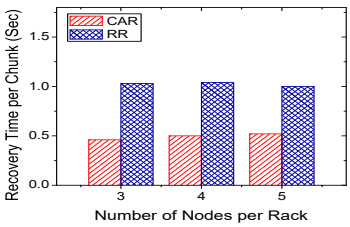
Figure 12(a) first gives the amount of caused cross-rack repair traffic when the number of data chunks (i.e., k) in a stripe varies. We make three observations. First, both CAR and RR incur more cross-rack repair traffic when the value of k becomes larger. The reason is that to repair a lost chunk, k surviving chunks are required in RS Codes. Second, CAR is more insensitive with the change of k when compared to RR. Third, CAR can still reduce the amount of cross-rack repair traffic when the value of k varies.

Figure 12(b) shows the time to repair a lost chunk when the value of k varies. We make two observations. First, both of CAR and RR incur more time to reconstruct a chunk when the value of k is larger, mainly because of the increased amount of cross-rack traffic during recovery. Second, CAR incurs less recovery time compared to RR. For example, when $k = 4$, CAR requires about 47.9% less time to repair a chunk when compared with RR.

Figure 13(a) shows the amount of cross-rack repair traffic when the number of parity chunks (i.e., m) in a stripe changes. We see that the selection of m in our evaluation does not have significant impact on the amount of cross-rack repair traffic for both RR and CAR. The reason is that the configuration of m does not affect the number of chunks to be read for recovery, even though it may increase the number of chunks in a stripe. Following the same reason, Figure 13(b) also indicates that the number of parity chunks has limited influence on the recovery time.



(a) Cross-rack repair traffic



(b) Recovery time

Fig. 14. Experiment 4: Impact of number of nodes per rack.

Experiment 4 (Impact of Number of Nodes Per Rack). We further study the impact of number of nodes per rack. We select the $(k = 4, m = 3)$ RS code and fix the chunk size as 4MB. The evaluated CFS in this test is constructed over three racks $\{A_1, A_2, A_3\}$ in Table 3, where each rack has three nodes. We then erase the data on a randomly selected node, vary the number of nodes in each rack from three to five, and measure the average amounts of cross-rack traffic and recovery time to repair a lost chunk. The evaluation results are shown in Figure 14.

Figure 14(a) shows that the amounts of cross-rack repair traffic to repair a lost chunk in both CAR and RR will not be significantly affected when the number of nodes per rack varies. The reason is that each rack always has the same number of nodes (i.e., from three to five) in each test, and this results in the same likelihood of placing a chunk in any one of the racks during the distribution of chunks. As a result, the number of chunks in each rack will not be affected even when the number of nodes per rack varies. Combined with the recovery principle of RR and CAR, their amounts of cross-rack repair traffic will not be influenced.

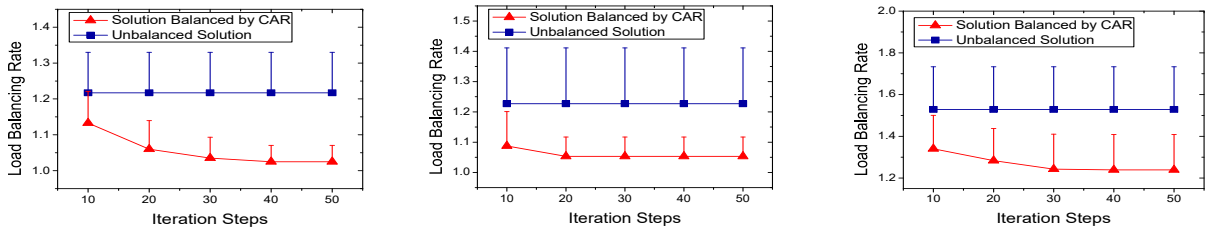
Figure 14(b) shows the recovery time of CAR and RR versus the number of nodes per rack. As the recovery time is closely related to the amount of cross-rack repair traffic, we observe that the recovery time of both CAR and RR will be stable when the number of nodes per rack changes.

Experiment 5 (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 15 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 15(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 resulting solution is closer to the minimum with the increase of iteration steps.

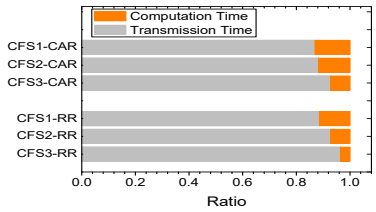
Experiment 6 (Computation Time and Transmission Time). We further provide a breakdown on the recovery time, in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TDSC.2017.2774299, IEEE Transactions on Dependable and Secure Computing

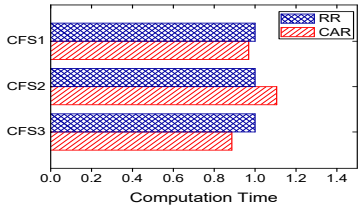


(a) CFS1 (10 nodes, $k = 4, m = 3$) (b) CFS2 (13 nodes, $k = 6, m = 3$) (c) CFS3 (20 nodes, $k = 10, m = 4$)

Fig. 15. Experiment 5: 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) Ratios of transmission time and computation time



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

Fig. 16. Evaluation of transmission time and computation time for recovering a lost chunk.

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. We fix the chunk size as 8MB.

Figure 16 presents the results. Figure 16(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 16(b) shows that the computation time of CAR normalized over that of RR. 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 only breaks down a decoding operation into multiple partial decoding operations.

6 Conclusions

This paper reconsiders the single failure recovery problem in a clustered file system (CFS) with over-subscribed cross-rack bandwidth, and propose CAR, a *cross-rack-aware recovery* algorithm. 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. Experimental results show that CAR can reduce both cross-rack data transmissions and the overall recovery time.

Acknowledgment

This work is supported by the National Natural Science Foundation of China (Grant No. 61602120, 61672159, 61232003, 61433008), the Technology Innovation Platform Project of Fujian Province (Grant No. 2014H2005), the Fujian Collaborative Innovation Center for Big Data Application in Governments, the Fujian Engineering Research Center of Big Data Analysis and Processing, and the Fujian Provincial Natural Science Foundation (Grant No. 2017J05102). This work is also supported by the Research Grants Council of Hong Kong (GRF 14216316 and CRF C7036-15G).

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] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010.

[3] S. Bhattacharya, S. Pratt, B. Pulavarty, and J. Morgan. Asynchronous i/o support in linux 2.5. In *Proceedings of the Linux Symposium*, pages 371–386, 2003.

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

[5] 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.

[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. Software engineering advice from building large-scale distributed systems. *CS295 Lecture at Stanford University, July*, 2007.

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

[10] 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.

[11] 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.

[12] 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.

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

[14] Y. Hu, P. P. Lee, and X. Zhang. Double regenerating codes for hierarchical data centers. In *Proc. of IEEE International Symposium on Information Theory*, 2016.

[15] Y. Hu, X. Li, M. Zhang, P. P. Lee, X. Zhang, P. Zhou, and D. Feng. Optimal repair layering for erasure-coded data centers: From theory to practice. *ACM Transactions on Storage*, 2017.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] R. Li, X. Li, P. P. Lee, and Q. Huang. Repair pipelining for erasure-coded storage. In *Proc. of USENIX ATC*, 2017.

[22] 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.

[23] S. Mitra, R. Panta, M.-R. Ra, and S. Bagchi. Partial-parallel-repair (ppr): a distributed technique for repairing erasure coded storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

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

[25] 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.

[26] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, Z. Wilcox-O'Hearn, et al. A performance evaluation and examination of open-source erasure coding libraries for storage. In *In Proc. of USENIX FAST*, 2009.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, and A. e. a. Dimakis. Xoring elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, 2013.

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

[33] 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.

[34] 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.

[35] 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.

[36] Z. Shen, J. Shu, and P. P. C. Lee. Reconsidering single failure recovery in clustered file systems. In *Proc. of IEEE/IFIP DSN*, 2016.

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

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

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

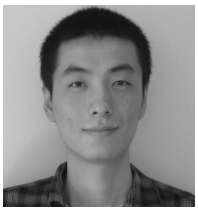
[40] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failure in rdp code storage systems. In *Proc. of ACM SIGMETRICS*, 2010.

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

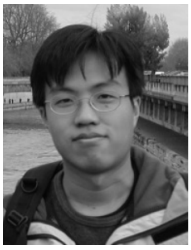
[42] S. Xu, R. Li, P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. Lui. Single disk failure recovery for x-code-based parallel storage systems. *IEEE Trans. on Computers*, 63(4):995–1007, 2014.

[43] 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.

[44] 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.



Zhirong Shen received the BS degree from University of Electronic Science and Technology of China, the Ph.D degree with Department of Computer Science and Technology at Tsinghua University in 2016. He is now a postdoctoral fellow at The Chinese University of Hong Kong. His current research interests include storage reliability and storage security.



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

Jiwu Shu received the Ph.D degree in computer science from Nanjing University in 1998, and finished the postdoctoral position research at Tsinghua University in 2000. Since then, he has been teaching at Tsinghua University. His current research interests include storage security and reliability, non-volatile memory based storage systems, and parallel and distributed computing. He is an senior member of the IEEE.

Wenzhong Guo received the BS and MS degrees in computer science, and the PhD degree in communication and information system from Fuzhou University, Fuzhou, China, in 2000, 2003, and 2010, respectively. He is currently a full professor with the College of Mathematics and Computer Science at Fuzhou University. His research interests include intelligent information processing, sensor networks, network computing, and network performance evaluation.