CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems

Runhui Li, Jian Lin, Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong
{*rhli, jlin, pclee*}*@cse.cuhk.edu.hk*

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

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

I. Introduction

To provide high storage capacity, large-scale distributed storage systems have been widely deployed in enterprises [2], [8]. In such systems, data is striped across multiple nodes (or servers) that are interconnected over a networked environment. Ensuring data availability in distributed storage systems is critical, since node failures are prevalent [8]. Data availability can be achieved via *erasure codes*, which encode original data and stripe encoded data across multiple nodes. Erasure codes can tolerate multiple failures and allow the original data to remain accessible by decoding the encoded data stored in other surviving nodes. Compared to replication, erasure codes have less storage overhead at the same fault tolerance.

In addition to tolerating failures, another crucial availability requirement is to *recover* any lost or unavailable data of failed nodes. To achieve high-performance recovery, one approach is to minimize the *recovery bandwidth* (i.e., the amount of data transfer over a network during recovery) based on *regenerating codes* [5], in which each surviving node encodes its stored data and sends encoded data for recovery. In the scenario where network capacity is limited, minimizing the recovery bandwidth can improve the overall recovery performance. In this work, we explore the feasibility of deploying regenerating codes in practical distributed storage systems.

However, most existing recovery approaches are designed to optimize *single failure* recovery. Although single failures are common, node failures are often correlated and co-occurring in practice, as reported in both clustered storage (e.g., [7], [25]) and wide-area storage (e.g., [3], [9], [18]). In addition, concurrent recovery is beneficial to delaying immediate recovery [1]. That is, we can perform recovery only when the number of failures exceeds a tolerable limit. This avoids unnecessary recovery should a failure be transient and the data be available shortly (e.g., after rebooting a failed node). Given the importance of concurrent recovery, we thus pose the following question: Can we achieve bandwidth saving, based on regenerating codes, in recovering a general number of failures including single and concurrent failures?

In this paper, we propose a complete system called CORE, which supports both single and concurrent failure recovery and aims to minimize the bandwidth of recovering a *general* number of failures. CORE augments existing optimal regenerating code constructions (e.g., [21], [30]), which are designed for single failure recovery, to also support concurrent failure recovery. A key feature of CORE is that it retains existing optimal regenerating code constructions and the underlying regenerating-coded data. That is, CORE adds a new recovery scheme atop existing regenerating codes. This paper makes the following contributions. We theoretically show that CORE achieves the minimum recovery bandwidth for a majority of concurrent failure patterns. We also propose extensions to CORE to achieve sub-optimal bandwidth saving even for the remaining concurrent failure patterns. We implement and experiment our CORE prototype on a Hadoop Distributed File System (HDFS) [29] testbed with up to 20 storage nodes. We show that compared to erasure codes, CORE achieves recovery throughput gains with up to $3.4\times$ for single failures and up to $2.3\times$ for concurrent failures.

The rest of the paper proceeds as follows. Section II first formulates our system model. Section III describes the design of CORE and presents our theoretical and analysis findings. Section IV presents experimental results. Section V reviews related work, and Section VI concludes this paper.

II. System Model

A. Basics

We first define the terminologies and notation. We consider a distributed storage system composed of a collection of *nodes*, each of which refers to a physical storage device. The storage system contains n nodes labeled by $N_0, N_1, \cdots, N_{n-1}$, in which k nodes (called *data nodes*) store the original (uncoded)

978-1-4799-0218-7/13/$31.00

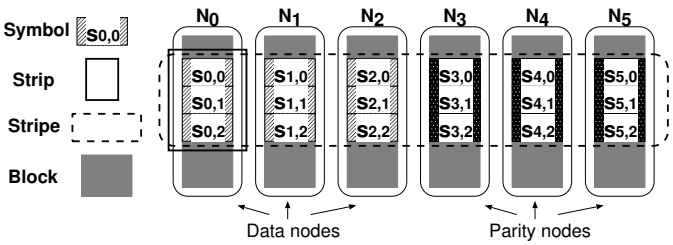


Fig. 1. Example of a distributed storage system, where $n = 6$, $k = 3$, and $r = 3$. We assume that nodes N_0, N_1, and N_2 are data nodes, while N_3, N_4, and N_5 are parity nodes. For load balancing, the identities of data and parities nodes are rotated across different blocks.

data and the remaining $n - k$ nodes (called *parity nodes*) store parity (coded) data. The coding structure is *systematic*, meaning that the original data is kept in storage.

Figure 1 shows an example of a distributed storage system. Each node stores a number of *blocks*. A block is the basic unit of read/write operations in a storage system. It is called a data block if it holds original data, or a parity block if it holds parity data. To store data/parity information, each block is partitioned into fixed-size *strips*, each of which contains r *symbols*. A symbol is the basic unit of encoding/decoding operations. A *stripe* is a collection of strips on k data nodes and the corresponding encoded strips on $n - k$ parity nodes.

Each stripe is independently encoded. Our discussion thus focuses on a single stripe and our recovery scheme will operate on a per-stripe basis. Let M be the total amount of original uncoded data stored in a stripe. Let $s_{i,j}$ be a stored symbol of node N_i at offset j in a stripe, where $i = 0, 1, \cdots, n-1$ and $j = 0, 1, \cdots r-1$. Note that our recovery scheme applies to the failures of both data and parity nodes. It treats each stored symbol $s_{i,j}$ the same way regardless of whether it is a data or parity symbol.

For data availability, we have the storage system employ an (n, k) code that is *maximum distance separable (MDS)*, meaning that the stored data of any k out of the n nodes can be used to reconstruct the original data. That is, an (n, k) MDS-coded storage system can tolerate any $n - k$ out of n concurrent failures. MDS codes also ensure optimal storage efficiency, such that each node stores $\frac{M}{k}$ units of data per stripe. Reed-Solomon (RS) codes [23] are a classical example of MDS codes. RS codes can be implemented with strip size $r = 1$ to minimize the generator matrix size.

B. Recovery

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

Our recovery builds on the *relayer* model, in which a relayer daemon coordinates the recovery operation. Figure 2 depicts the relayer model. During recovery, each surviving node performs two steps: (i) *I/O*: it reads its stored data,

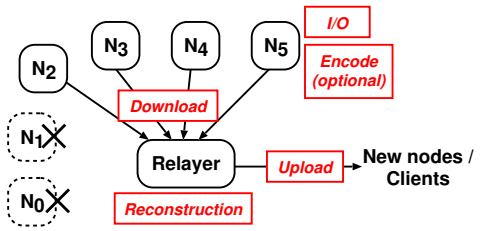


Fig. 2. Recovering nodes N_0 and N_1 using the relayer model.

and (ii) *encode* (for regenerating codes only): it combines the stored data into some linear combinations. The relayer daemon performs three steps: (i) *download*: it downloads the data from some other surviving nodes, (ii) *reconstruction*: it reconstructs the lost data, and (iii) *upload*: it uploads the reconstructed data to the new nodes (for recovery from permanent failures) or to the client who requests the data (for degraded reads). We assume that the relayer is reliable during the recovery process.

We argue that the relayer model can be easily fit into practical distributed storage systems. In the case of recovering permanent failures, we can deploy the relayer daemon in different ways, such as in one of the new storage nodes that reconstructs all lost data, in every storage node that reconstructs a subset of lost data, or in separate servers that run outside the storage system.

To improve the recovery performance of a distributed storage system, we need to minimize the amount of data transferred over the network. If the number of failed nodes is small, the amount of data being downloaded from the surviving nodes is larger than the amount of reconstructed data being uploaded to new nodes (or clients). If we pipeline the download and upload steps (see our technical report [17]), then the download step becomes the bottleneck. Thus, we focus on optimizing the download step in recovery. We define the *recovery bandwidth* as the total amount of data being downloaded per stripe from the surviving nodes to the relayer during recovery. Our goal is to minimize the recovery bandwidth.

C. Regenerating Codes

We consider a special class of codes called *regenerating codes* [5], which enable the relayer to transfer less than the amount of original data being stored. During recovery, surviving nodes send encoded symbols that are computed by the linear combinations of their stored symbols, and then the encoded symbols are used to reconstruct the lost data. It is shown that regenerating codes lie on an optimal tradeoff curve between storage cost and recovery bandwidth [5]. In this work, we focus on one optimal extreme point called the *minimum storage regenerating (MSR)* codes, in which each node stores the minimum amount of data on the tradeoff curve. MSR codes have the same optimal storage efficiency as MDS erasure codes such as RS codes.

Existing optimal MSR codes are designed for recovering a single failure, as described below. First, the strip size has $r = n - k$ symbols to achieve the minimum possible bandwidth. During recovery, the relayer downloads one *encoded*

symbol from each of the $n-1$ surviving nodes[1]. Let $e_{i,i'}$ be the encoded symbol downloaded from node N_i and used to reconstruct data for the failed node $N_{i'}$. Each encoded symbol $e_{i,i'}$ is a function of the symbols $s_{i,0}, s_{i,1}, \cdots, s_{i,r-1}$ stored in the surviving node N_i, and has the same size as each stored symbol. Using the encoded symbols, the relayer reconstructs the lost symbols of the failed node $N_{i'}$. The theoretical lower bound of recovery bandwidth for MSR is [5]:

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

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

III. Design of CORE

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

In this section, we first describe the baseline approach of CORE, in which we extend the existing optimal solution of single failure recovery to support concurrent failure recovery (Section III-A). We note that the baseline approach of CORE is not applicable in a small proportion of failure patterns, so we propose a simple extension that still provides bandwidth reduction for such cases (Section III-B). We present theoretical results that show that CORE can reach the optimal point in a majority of failure patterns (Section III-C). Finally, we analyze the recovery bandwidth saving of CORE (Section III-D).

A. Baseline Approach of CORE

Building blocks. We refer readers to our technical report [17] for the background details of the MSR code constructions. Our observation is that any optimal MSR code construction can be defined by two functions. Let $\mathsf{Enc}_{i,j}$ be the encoding function that is called by node N_i to generate an encoded symbol for the failed node N_j using the $r = n-k$ stored symbols in node N_i as inputs; let Rec_i be the reconstruction function that returns the set of $n-k$ stored symbols of a failed node N_i using the encoded symbols from the other $n-1$ surviving nodes as inputs. Both Enc and Rec define the operations of linear combinations of the stored symbols $s_{i,j}$'s, depending on the specific code construction.

[1]There are MSR code constructions (e.g., [21], [30]) that can download encoded symbols from less than $n-1$ surviving nodes at the expense of higher recovery bandwidth. In this work, we only focus on the case where $n-1$ surviving nodes are connected.

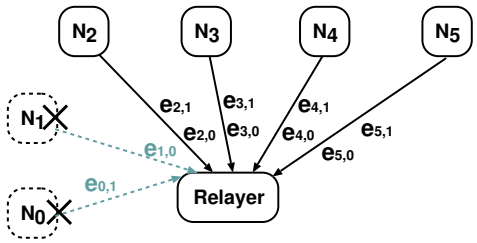


Fig. 3. An example of how the relayer downloads real and virtual symbols for a (6,3) code when there are two failed nodes N_0 and N_1. Here, $e_{1,0}$ and $e_{0,1}$ are the virtual symbols.

CORE works for *any* construction of optimal MSR codes, as long as the functions Enc and Rec are well-defined. The two functions Enc and Rec form the building blocks of CORE. CORE can build on existing optimal MSR code constructions including Interference Alignment (IA) codes [30] and Product-Matrix (PM) codes [21]. Note that both IA and PM codes have parameter constraints. IA codes require $n \geq 2k$, and PM codes require $n \geq 2k-1$. In this work, we mainly focus on the double redundancy $n = 2k$, which is also considered in state-of-the-art distributed storage systems such as OceanStore [16] and CFS [4].

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

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

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

The encoded symbol $e_{1,0}$ is computed by encoding the stored symbols $s_{1,0}$, $s_{1,1}$, and $s_{1,2}$, all of which can be reconstructed from other encoded symbols $e_{0,1}$, $e_{2,1}$, $e_{3,1}$, $e_{4,1}$, and $e_{5,1}$ based on single failure recovery. Thus, $e_{1,0}$ can be expressed as a function of encoded symbols. The encoded symbol $e_{0,1}$ is expressed in a similar way. Now, we have two equations

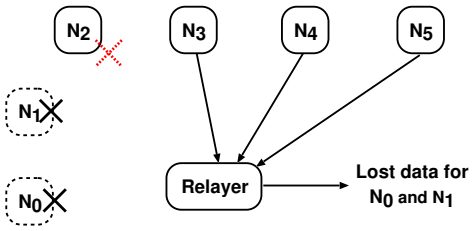


Fig. 4. An example of using a virtual failure pattern for a (6,3) code. If the original failure pattern $\{N_0, N_1\}$ is bad, then we can instead recover the virtual failure pattern $\{N_0, N_1, N_2\}$ and only download encoded symbols from nodes N_3, N_4, N_5.

with two unknowns $e_{1,0}$ and $e_{0,1}$. If these two equations are linearly independent, we can solve for $e_{1,0}$ and $e_{0,1}$. Then we can apply Rec_i to reconstruct the lost stored symbols of node N_i. In general, to recover t failed nodes, we have a total of $t(t-1)$ virtual symbols. We can compose $t(t-1)$ equations based on the above idea. If these $t(t-1)$ equations are linearly independent, we can solve for the virtual symbols. A subtle issue is that the system of equations may be unsolvable. We explain how we generalize our baseline approach for such an issue in the next subsection.

B. Recovering Any Failure Pattern

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

As shown in our technical report [17], the proportion of bad failure patterns is in general very small, with at most 0.9% and 1.6% for IA and PM codes, respectively. Also, for some sets of parameters, we do not find any bad failure patterns. Nevertheless, we would like to reduce the recovery bandwidth for such bad failure patterns even though they are rare.

We now extend our baseline approach of CORE to deal with the bad failure patterns, with an objective of reducing the recovery bandwidth over the conventional recovery approach. For a bad failure pattern $\mathcal{F}$, we include one additional surviving node and form a *virtual failure pattern* $\mathcal{F}'$, such that $\mathcal{F} \subset \mathcal{F}'$ and $|\mathcal{F}'| = |\mathcal{F}| + 1 = t + 1$. Then the relayer downloads the data from the $n-t-1$ nodes outside $\mathcal{F}'$ needed for reconstructing the lost data of $\mathcal{F}'$, although actually only the lost data of $\mathcal{F}$ needs to be reconstructed. Figure 4 shows an example of how we use a virtual failure pattern for recovery. If $\mathcal{F}'$ is still a bad failure pattern, then we include an additional surviving node into $\mathcal{F}'$, and repeat until a good failure pattern is found. Note that the size of $\mathcal{F}'$ must be upper-bounded by $n-k$, as we can always connect to k surviving nodes to reconstruct the original data due to the MDS code property.

C. Theoretical Results

We present two theorems. The first one shows the lower bound of recovery bandwidth. The second one shows that CORE achieves the lower bound for good failure patterns. Since most failure patterns are good, we conclude that CORE minimizes recovery bandwidth for a majority of failure patterns. The proofs are in [17].

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

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

□

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

D. Analysis of Bandwidth Saving

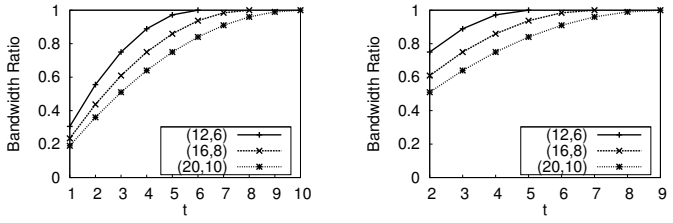
We now study the bandwidth saving of CORE over conventional recovery. We compute the bandwidth ratio, defined as the ratio of recovery bandwidth of CORE to that of conventional recovery. We vary (n, k) and the number t of failed nodes to be recovered.

We first consider good failure patterns. Figure 5(a) shows the bandwidth ratio. We observe that CORE achieves bandwidth saving in both single and concurrent failures. For single failures (i.e., $t = 1$), CORE directly benefits from existing regenerating codes, and saves the recovery bandwidth by 70-80%. For concurrent failures (i.e., $t > 1$), CORE also shows the bandwidth saving, for example by 44-64%, 25-49%, and 11-36% for $t = 2$, $t = 3$ and $t = 4$, respectively. The bandwidth saving decreases as t increases, since more lost data needs to be reconstructed and we need to retrieve nearly the amount of original data stored. On the other hand, the bandwidth saving increases with the values of (n, k). For example, the saving is 36-64% in (20,10) when $2 \leq t \leq 4$.

We now study how CORE performs for bad failure patterns. Recall from Section III-B for each bad failure pattern $\mathcal{F}$, CORE forms a virtual failure pattern $\mathcal{F}'$ that is a good failure pattern. We compute the recovery bandwidth for $\mathcal{F}'$ based on our theoretical results in Section III-C. Figure 5(b) shows the bandwidth ratio. We find that in all cases we consider, it suffices to add one surviving node into $\mathcal{F}'$ (i.e., $|\mathcal{F}'| = |\mathcal{F}| + 1$) and obtain a good failure pattern. Thus, the recovery bandwidth of CORE for a bad t-failure pattern is always equivalent to that for a good $(t+1)$-failure pattern. From the figure, we still see bandwidth saving of CORE over conventional recovery. For example, the saving is 25-49% in (20,10) when $2 \leq t \leq 4$.

IV. Prototype Experiments

We implement CORE on Hadoop Distributed File System (HDFS) [29] with its erasure-coded module HDFS-RAID [10]. We experiment CORE on an HDFS testbed with one NameNode and up to 20 DataNodes being used. Each node runs on a quad-core PC equipped with an Intel Core i5-2400 3.10GHz CPU, 8GB RAM, and a Seagate ST31000524AS 7200RPM 1TB SATA harddisk. All machines are equipped with a 1Gb/s Ethernet card and interconnected over a 1Gb/s Ethernet switch.



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

Fig. 5. Ratio of recovery bandwidth of CORE to that of conventional recovery.

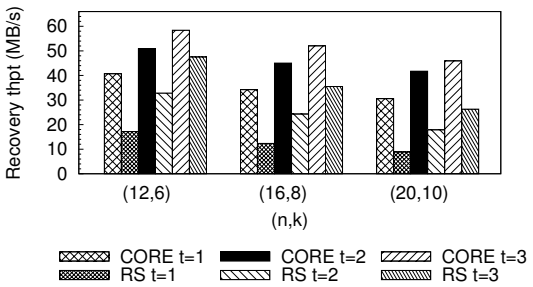


Fig. 6. Recovery throughput.

They all run Linux Ubuntu 12.04. We compare RS codes [23], which use conventional recovery, and CORE, which builds on IA codes [30]. Both codes are implemented in C++ using the Jerasure library [20] and compiled with GCC 4.6.3 with the -O3 option. Our results are averaged over five runs.

We evaluate the recovery performance. For a given (n, k), we configure our HDFS testbed with n DataNodes, one of which also deploys the RaidNode for striping the encoded data. We prepare a kGB of original data as our input. By our observation, the input size is large enough to give a steady throughput. HDFS first stores the file with the default 3-replication scheme. Then the RaidNode stripes the replica data into encoded data using either RS codes or IA codes. The encoded data is stored in n DataNodes. We rotate node identities when we place the blocks so that the parity blocks are evenly distributed across different DataNodes to achieve load balancing. We fix the symbol size at 8KB. We use the default HDFS block size at 64MB, but for some (n, k), we alter the block size slightly to make it a multiple of the strip size (which is $(n - k)\times$8KB) for IA codes.

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

Figure 6 shows the recovery throughput results. Both CORE and RS codes see higher throughput for larger t as more lost blocks are recovered. Overall, CORE shows significantly higher throughput than RS codes. The throughput gain is the highest in (20,10). For example, for single failures, the gain is 3.45×; for concurrent failures, the gains are 2.33× and 1.75× for $t = 2$ and $t = 3$, respectively.

Our experimental results are fairly consistent with our analytical results in Section III-D. For example, in (20,10), the ratio of the reconstruction bandwidth of CORE to that of erasure codes for $t = 2$ and $t = 3$ are 0.36 and 0.51, respectively (see Figure 5(a)). These results translate to the recovery throughput gains of CORE at 2.78× and 1.96×, respectively. Our experimental results show slightly less gains, mainly due to disk I/O and encoding/decoding overheads that are not captured in the recovery bandwidth. We present more evaluation results in our technical report [17].

V. Related Work

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

Minimizing I/Os. Several studies [31]–[33] focus on minimizing I/Os required for recovering a single failure in RAID-6 codes. Their approaches mainly focus on a disk array system. Authors of [15] show that finding the optimal recovery solution for arbitrary erasure codes is NP-hard. Authors of [34], [35] propose greedy heuristics to speed up the search of solutions for single failure recovery. Authors of [6], [13], [19], [24][2] propose local recovery codes that reduce the bandwidth and I/O when reconstructing a lost data fragment. They evaluate the codes atop a cloud storage system simulator (e.g., in [19]), Azure Storage (e.g., in [13]), and HDFS (e.g., in [6], [24]). It is worth noting that the local recovery codes are non-MDS codes with additional parities added to storage, so as to trade for better recovery performance. On the other hand, our work builds on MSR codes that have the same optimal storage efficiency as MDS codes.

Minimizing recovery bandwidth. Regenerating codes [5], [21], [22], [26], [30] minimize the recovery bandwidth for a single failure in a distributed storage system. In contrast with the above solutions that minimize I/Os, most regenerating codes typically read all stored data to generate encoded data during recovery. Authors of [11] implement and experiment non-systematic minimum storage regenerating codes in a cloud storage system prototype, without requiring storage nodes to perform encoding operations (i.e., uncoded recovery).

Cooperative recovery. Several theoretical studies (e.g., [12], [14], [27], [28]) address concurrent failure recovery based on regenerating codes, and they focus on recovery of lost data on new nodes. They all consider a *cooperative model*, in which the new nodes exchange among themselves their data being read from surviving nodes during recovery. Authors of [12], [14] prove that the cooperative model achieves the same optimal recovery bandwidth as ours, but they do not provide explicit constructions of regenerating codes that achieve the optimal point. Authors of [27], [28] provide such explicit

[2]Although the proposed scheme of [6] is also called CORE, it refers to Cross Object Redundancy and addresses a different problem from ours.

implementations, but they focus on limited parameters and the resulting implementations do not provide any bandwidth saving over erasure codes. A drawback of the cooperative model requires coordination among the new nodes to perform recovery, and its implementation complexities are unknown.

VI. Conclusions

We explore the use of regenerating codes to provide fault-tolerant storage and minimize the bandwidth of data transfer during recovery. We propose a system CORE, which generalizes existing optimal single-failure-based regenerating codes to support the recovery of both single and concurrent failures. We theoretically show that CORE minimizes the reconstruction bandwidth in most concurrent failure patterns. To demonstrate, we prototype CORE as a layer atop Hadoop HDFS, and show via testbed experiments that we can speed up the recovery operation. The source code of CORE is available for download at **http://ansrlab.cse.cuhk.edu.hk/software/core**.

Acknowledgments

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

References

[1] R. Bhagwan, K. Tati, Y. Cheng, S. Savage, and G. Voelker. Total Recall: System Support for Automated Availability Management. In *Proc. of USENIX NSDI*, Oct 2004.

[2] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *Proc. of ACM SOSP*, Oct 2011.

[3] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient Replica Maintenance for Distributed Storage Systems. In *Proc. of USENIX NSDI*, May 2006.

[4] F. Dabek, M. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, Dec 2001.

[5] A. Dimakis, P. Godfrey, Y. Wu, M. Wainwright, and K. Ramchandran. Network Coding for Distributed Storage Systems. *IEEE Trans. on Information Theory*, 56(9):4539–4551, Sep 2010.

[6] K. Esmaili, P. Lluis, and A. Datta. The CORE Storage Primitive: Cross-Object Redundancy for Efficient Data Repair & Access in Erasure Coded Storage. *arXiv*, preprint arXiv:1302.5192, 2013.

[7] D. Ford, F. Labelle, F. I. Popovici, M. Stokel, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proc. of USENIX OSDI*, Oct 2010.

[8] S. Ghemawat, H. Gobioff, and S. Leung. The Google File System. In *Proc. of ACM SOSP*, Dec 2003.

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

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

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

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

[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*, Jun 2012.

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

[15] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *Proc. of USENIX FAST*, Feb 2012.

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

[17] R. Li, J. Lin, and P. Lee. CORE: Augmenting Regenerating-Coding-Based Recovery for Single and Concurrent Failures in Distributed Storage Systems. *arXiv*, preprint arXiv:1302.3344, 2013.

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

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

[20] J. Plank, J. Luo, C. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proc. of USENIX FAST*, Feb 2009.

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

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

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

[24] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur. XORing Elephants: Novel Erasure Codes for Big Data. *Proceedings of the VLDB Endowment (to appear)*, 2013.

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

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

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

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

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

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

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

[32] L. Xiang, Y. Xu, J. Lui, Q. Chang, Y. Pan, and R. Li. A Hybrid Approach to Failed Disk Recovery Using RAID-6 Codes: Algorithms and Performance Evaluation. *ACM Trans. on Storage*, 7(3):11, Oct 2011.

[33] 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*, To appear.

[34] Y. Zhu, 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*, Apr 2012.

[35] Y. Zhu, 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 DSN*, Jun 2012.