CORE: Cross-Object Redundancy for Efficient Data Repair in Storage Systems

Kyumars Sheykh Esmaili
Nanyang Technological University
Singapore 639798
kyumarss@ntu.edu.sg

Lluis Pamies-Juarez
Nanyang Technological University
Singapore 639798
lpjuarez@ntu.edu.sg

Anwitaman Datta
Nanyang Technological University
Singapore 639798
anwitaman@ntu.edu.sg

***Abstract*—Erasure codes are an integral part of many distributed storage systems aimed at Big Data, since they provide high fault-tolerance for low overheads. However, traditional erasure codes are inefficient on replenishing lost data (vital for long term resilience) and on reading stored data in degraded environments (when nodes might be unavailable). Consequently, novel codes optimized to cope with distributed storage system nuances are vigorously being researched.**

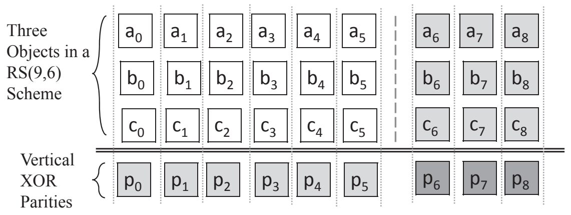
In this paper, we take an engineering alternative, exploring the use of simple and mature techniques – juxtaposing a standard erasure code with RAID-4 like parity to realize cross object redundancy (CORE), and integrate it with HDFS. We benchmark the implementation in a proprietary cluster and in EC2. Our experiments show that for an extra 20% storage overhead (compared to traditional erasure codes) CORE yields up to 58% saving in bandwidth and is up to 76% faster while recovering a single failed node. The gains are respectively 16% and 64% for double node failures.

I. Introduction

In order to meet the conflicting needs of high fault-tolerance and low storage overhead, erasure codes are increasingly being embraced for distributed storage systems aimed to store high volumes of data. Traditional erasure codes have mostly been designed to optimize the performance of communication-centric applications, and are not necessarily amenable to the needs of storage systems. Some such desirable properties include efficient replenishment of lost redundancy (repair) following the failure of some system components; and efficient access of data while the system is yet to complete remedial actions following such failures (degraded reads/access). To that end, there has been tremendous interest in both coding theory and storage systems research communities to build new erasure codes with good repairability properties, as well as building robust storage systems leveraging on the novel codes (for instance, Windows Azure Storage using Local Reconstruction Codes). In this paper we explore an alternate design, looking at an instance of product codes [1]. A traditional erasure code is first applied on individual data objects, followed by the creation of RAID-4 like parity over erasure encoded pieces of different objects, creating cross-object redundancy. This results in high fault tolerance (provided by the traditional code) and cheap repairs (provided by the parity code). The approach is simple, and based on mature techniques that have long been used as stand-alone approaches (these are desirable for practical and implementation considerations), yet it achieves very good (less communication and computation) repairability and degraded data access under many fault-conditions. We accordingly build the *CORE storage primitive* as a general purpose, block level, fault-tolerant, data storage layer that can be readily integrated into distributed file systems relying on an underlying block level storage, providing significant performance boost. We integrate CORE into Hadoop Distributed File System (HDFS), and benchmark it over a wide range of system configurations, comparing it with state-of-the-art alternatives [2], [3] to demonstrate its efficacy. CORE builds upon our recent theoretical study [4] where we made a simple observation - by introducing a RAID-4 like parity over a small set of erasure encoded pieces, it is possible to achieve significant reduction in the expected cost to repair lost redundancy. Moreover, since these extra parities are relatively-small, the fault tolerance of the resulting system is only marginally lower than what is achieved with optimal (maximum distance separable, or MDS) codes –i.e. Reed-Solomon codes. This suboptimal fault-tolerance is expected from any code aiming to reduce the costs of repairs. Gopalan et al. studies the involved trade-offs [5] .

In Figure 1 we show an example to elaborate the basic idea on which CORE is built. Consider three objects (a, b, c), each comprising of 6 blocks. Each of these objects are first individually encoded using a (9,6) Reed-Solomon code. Note that each row represents an object along with its three parity blocks (depicted in gray). Additionally, a simple parity check (i.e. an XOR) is computed over each column's blocks and thereby a new row is added at the bottom of the matrix. In this example, this extra row increases the storage overhead by 33%. In the general case, the overhead is $1/t$ more than MDS codes, where t is the number of objects cross-coded together. As shown in our technical report [6], CORE's parameters can be adjusted to operate at reasonable overheads, even while achieving very good fault-tolerance as well as repairability. In particular, for equivalent storage overhead, CORE's performance benefit is significantly better than the state-of-the-art Locally Reconstruction Codes used in Azure [3], while, CORE achieves fault-tolerance (ignoring

This work is supported by Singapore's A*Star Grant 102 158 0038 & National Research Foundation Grant NRF-CRP2-2007-03.



Three Objects in a RS(9,6) Scheme
Vertical XOR Parities
a_0 a_1 a_2 a_3 a_4 a_5 a_6 a_7 a_8
b_0 b_1 b_2 b_3 b_4 b_5 b_6 b_7 b_8
c_0 c_1 c_2 c_3 c_4 c_5 c_6 c_7 c_8
p_0 p_1 p_2 p_3 p_4 p_5 p_6 p_7 p_8

Figure 1: An example illustrating the basic idea of CORE

repairs) comparable to optimal MDS erasure codes for an acceptable 20% more storage overhead.

The main advantage that the vertical parities introduce in CORE is the increased efficiency of repairs. Fewer blocks are needed to carry out a repair. Furthermore, repair related computation is cheap (a simple XOR operation, compared to the expensive RS decoding procedure). In the example of Figure 1, repairing any single failure would require XORing 3 blocks. Apart from the repairability benefits, CORE's vertical parities also improve fault tolerance. For instance, in Figure 1, an object (i.e. a row) with more than three failures can be still recovered with the help of vertical parities. This improvement is however, not optimal in terms of the additional storage overhead, as the primary purpose of the vertical parities is to enhance the repairability aspect. The low repair cost in CORE also naturally translates into better degraded reads.

The main contributions of this work are as follows:

- Design and implementation of a general purpose, (efficiently) repairable block level storage primitive CORE, and its integration with a popular distributed file system, HDFS (our implementation is available at [7]).
- In the process, we identify a few ways to optimize the existing HDFS-RAID [2] implementation, on which we build CORE.
- We design novel algorithms to understand the failure patterns and adaptively exploit the better repair flexibility afforded by CORE's code design, in order to achieve fast and cheap repairs.

To repair single failures, CORE consumes 50% less bandwidth and is between 43% to 76% faster compared the classic erasure code. In case of double failures and in the worst case scenario –when both failed blocks belong to the same file– it consumes 16% less bandwidth and is 13% to 59% faster. We thus hope that CORE's design and analysis is not only academically interesting, but the performance boost it achieves despite a very simple design makes it a serious candidate for wide-scale adoption. We have also carried out analytical study of many more complex failure patterns, but we exclude the theoretical results due to space constraint, and they can be found in an extended version [6].

II. Related Work

Erasure codes have long been explored as a storage efficient alternative to replication for achieving fault-tolerance [8] in the peer-to-peer (P2P) systems literature, and have led to numerous prototypes, e.g., OceanStore [9] and TotalRecall [10] to name a few. In recent years erasure codes have gained traction [11] even in main-stream storage technologies such as RAID [12]. The ideas from RAID systems are in turn permeating to Cloud settings [13], [14], and erasure codes have become an integral part of many proprietary file systems used in data-centers [15], [16], as well as open-source variants [2].

With the proliferation of erasure codes in storage-centric applications, there has been a corresponding rise in the exploration of novel erasure codes which cater to the nuances of distributed storage systems. Specific aspects that have been investigated in designing such new coding techniques include: (i) *efficient degraded data access* [3], [17], (ii) *good repairability* [18], [19] by either combining standard codes [4], [20], [21], applying network coding techniques [19], [22], [23], or designing completely new codes with lower repair fan-in [24]–[27], and (iii) *fast creation* of erasure coded redundancy [28], [29].

Despite the plethora of works investigating novel erasure codes, most existing distributed file systems using erasure codes do so by adapting traditional erasure codes. Microsoft's Windows Azure Storage [15] is a prominent exception which uses an optimized version of Pyramid codes [17] called Local Reconstruction Code (LRC) [3]. Some recent academic prototypes - NCFS [30] and [31][1] likewise explore the feasibility of applying network coding techniques for repairing lost data. The latter systems do not address the issue of degraded reads. In contrast to these systems based on proprietary and novel erasure coding techniques with significant system design complexity, CORE combines two mature techniques (standard erasure codes and RAID-4 like parity) while achieving very good repairability and degraded read performance. This makes CORE suitable for ready integration with many block based storage/file systems, and its simple design makes it amenable to third party reimplementations.

III. Background

We next provide some background on what erasure codes are and how they are used in distributed storage systems, followed by a discussion on local repairability which has come to the fore in the design of novel storage-centric erasure codes. Finally, we discuss the code used in Azure system, which, to the best of our knowledge, was the first deployment of repairable codes in a large-scale commercial cloud storage system.

[1]Coincidentally, [31] uses the same name, CORE, for collaborative regeneration.

A. Classic Erasure Codes

Traditionally, large data objects have been stored by splitting them into blocks of (say) size q bits, which are then replicated across multiple storage nodes. In contrast, an (n, k) erasure code takes k different data blocks of size q, and computes $m = n - k$ parity blocks of the same size, each to be stored in a different storage node. Then, in the event of disk failures, the k original blocks can be reconstructed by collecting and decoding a subset of $k' \geq k$ blocks out of the total n stored blocks.

Consider that the vector $\mathbf{o} = (o_1, \ldots, o_k)$ denotes a data object composed of k blocks of q bits each. That is, each block o_i is a string of q bits. The encoding operations are performed using finite field arithmetic where the two bits $\{0, 1\}$ form a finite field $\mathbb{F}_2$ of two elements, while o_i likewise belongs to the binary extension field $\mathbb{F}_{2^q}$ containing 2^q elements. Then, the encoding of the object $\mathbf{o}$ is a linear transformation defined by a $k \times n$ generator matrix G such that we can obtain an n-dimensional codeword $\mathbf{c} = (c_1, \ldots, c_n)$ of size $n \times q$ bits by applying the linear transformation $\mathbf{c} = \mathbf{o} \cdot G$. A code with such a generator matrix G is usually referred to as an (n, k)-code. When the generator matrix G has the form $G = [I_k, G']$ where I_k is the identity matrix and G' is a $k \times m$ matrix ($m = n - k$), the codeword $\mathbf{c}$ becomes $\mathbf{c} = [\mathbf{o}, \mathbf{p}]$ where $\mathbf{o}$ is the original object, and $\mathbf{p}$ is a parity vector containing $m \times q$ parity bits. The code is then said to be *systematic*, in which case the k parts of the original object remain unaltered after the coding process. We want to note that the main advantage of systematic codes is that the original data $\mathbf{o}$ can be accessed without requiring a decoding process, by just reading the systematic blocks of $\mathbf{c}$.

The above encoding process stretches the original data by a factor of n/k (ratio known as the *stretch factor*), occupying n/k times more storage space than the size of the original object. By choosing a suitable code with a stretch factor satisfying $n/k < r$, significant storage space savings can be achieved in comparison to a system using r replicas. Finally, an *optimal erasure code* in terms of the trade-off between storage overhead and fault tolerance is called a *maximum distance separable* (MDS) code, and has the property that the original object $\mathbf{o}$ can be reconstructed from any k out of the total $n = k+m$ stored blocks (i.e., $k' = k$), tolerating the loss of any arbitrary $m = n - k$ blocks. The fault-tolerance of MDS erasure codes has been previously analyzed and compared with replication [8], [10], providing guidelines to choose suitable code parameters n and k for a desired level of resilience under an expected level of failures of individual storage nodes.

B. Locally Repairable Codes

A critical drawback of MDS codes is their high reconstruction cost. Repairing/reading a single failed block requires to download an amount of information equivalent to the size of the whole data object $\mathbf{o}$, which is k times larger than the amount of data being repaired/read.

Since repairs and degraded reads are frequent in storage systems, several recent works [3], [17], [24]–[26] have looked at reducing the number of blocks needed to carry out the repair/reconstruction of an inaccessible block (which is needed for both repair and access). Such a property is achieved by introducing 'local dependencies' among encoded blocks, and can be called repair locality.

Local repairability is achieved when a block c_i can be expressed as a linear combination of d ($d < k$) other blocks, $c_i = \alpha_1 c'_1 + \alpha_2 c'_2 + \cdots + \alpha_d c'_d$, where $c'_j \in \mathbf{c}$ s.t. $c'_j \neq c_i$, and $\alpha_j \in \mathbb{F}_{2^q}$ for all $j = 1, \ldots d$. This local repairability property allows to reduce the number of blocks accessed and transferred during degraded reads or repairs from k to d, where d can be as small as $d = 2$ [24], [25]. Unfortunately, achieving such code locality leads to poorer fault-tolerance for a given storage overhead in comparison to MDS codes [5]. Hence, the design of such codes poses a trade-off between three important desirable system properties: (i) high fault-tolerance, (ii) low storage overhead, and (iii) efficient repairs and degraded reads. For example, Pyramid codes [17] (the code behind Azure) were not originally conceived for efficient repairs per se, but to provide degraded read capabilities. In this case the code cannot obtain efficient repairs for all encoded blocks.

IV. Cross-Object Redundancy

We next explore how *product codes* [1] can achieve good repairability without compromising either the degraded read performance or the fault-tolerance of the code. Specifically, by combining a long and a short linear erasure code, we realize a product code with high fault tolerance (mainly provided by the long code) and high repair locality (provided by the short code). This is achieved by encoding multiple already-encoded objects together (or *cross-object* encoding), thus reusing existing encoding/decoding/repair mechanisms already deployed in a distributed storage system, facilitating an organic integration of the approach.

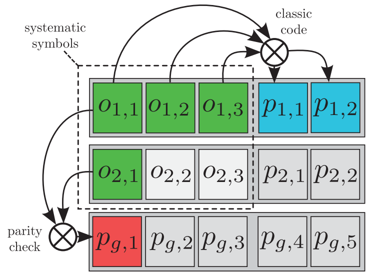
Example 1: Suppose that we have two different data objects $\mathbf{o}_1 = (o_{11}, o_{12}, o_{13})$ and $\mathbf{o}_2 = (o_{21}, o_{22}, o_{23})$ to be encoded with a (5,3) systematic MDS erasure code (with a generator matrix $G_{\mathbf{o}}$). Then, we obtain the codewords:

$$\mathbf{c}_1 = \mathbf{o}_1 \cdot G_{\mathbf{o}} = (o_{11}, o_{12}, o_{13}, p_{11}, p_{12}),$$
$$\mathbf{c}_2 = \mathbf{o}_2 \cdot G_{\mathbf{o}} = (o_{21}, o_{22}, o_{23}, p_{21}, p_{22}).$$

By grouping symbols from $\mathbf{c_1}$ and $\mathbf{c_2}$ in a per-column basis, we obtain the set of vectors

$$\mathcal{P} = \{(o_{11}, o_{21}), (o_{12}, o_{22}), (o_{13}, o_{23}), (p_{11}, p_{21}), (p_{12}, p_{22})\}.$$

We encode then each vector $x_i \in \mathcal{P}$ (cross-object encoding) with a (3,2) systematic code (a simple parity check code, or SPC), with generator matrix $G_{\mathbf{g}} = [I_2, \mathbf{1}_2]$), where I_2 is the identity matrix, and $\mathbf{1}_2$ is a vector with two ones. For



systematic symbols
classic code
parity check
$o_{1,1}$ $o_{1,2}$ $o_{1,3}$ $p_{1,1}$ $p_{1,2}$
$o_{2,1}$ $o_{2,2}$ $o_{2,3}$ $p_{2,1}$ $p_{2,2}$
$p_{g,1}$ $p_{g,2}$ $p_{g,3}$ $p_{g,4}$ $p_{g,5}$

Figure 2: Example of a simple product code. The blue parity blocks are generated using a horizontal (5,3) MDS code whereas the red block is a simple parity check of the column (or a (3,2) code).

each $x_i \in \mathcal{P}$ we obtain $\mathbf{p}_{g,i} = x_i \cdot G_\mathbf{g} = [x_i, \mathbf{p}_{g,i}]$, where $\mathbf{p}_{g,i} = (\sum x_i)$. The vector with all the cross-object parity blocks, $\mathbf{p}_g = (\mathbf{p}_{g,1}, \dots, \mathbf{p}_{g,5})$, contains:

$$\mathbf{p}_g = (o_{11}+o_{21},\ o_{12}+o_{22}, o_{13}+o_{23},\ p_{11}+p_{21},\ p_{12}+p_{22}).$$

In Fig. 2 we depict this two-phase encoding process. Note that $\mathbf{p}_g$ can be viewed as the Reed Solomon encoding of the respective parities of the systematic symbols. We refer to a code with a generator matrix G that takes a composed data object $\mathbf{o} = (\mathbf{o}_1, \mathbf{o}_2)$ and encodes it to a codeword $\mathbf{c} = \mathbf{o} \cdot G = [\mathbf{c}_1, \mathbf{c}_2, \mathbf{p}_g]$, as the product code of $G_\mathbf{g}$ and $G_\mathbf{o}$. It is easy to see how this example product code repairs any single missing block by using the outer erasure code $G_\mathbf{g}$, e.g., we can repair $o_{1,1}$ using $o_{1,1} = o_{2,1} + p_{g,1}$. In addition, in case of more than one failure per "column", the code still has the opportunity to repair up to two failures per codeword $\mathbf{c}_i$, and up to two failures within the extra parity vector $\mathbf{p}_g$.

Definition of CORE's Product Code: Let $G_\mathbf{c}$ and $G_\mathbf{o}$ respectively be the generator matrices of an (n_c, k_c) and an (n_o, k_o) code. Then, the product code of $G_\mathbf{c}$ and $G_\mathbf{o}$ is a $(n_c n_o, k_c k_o)$ linear code with generator matrix $G = G_\mathbf{c} \otimes G_\mathbf{o}$, where the operator $\otimes$ represents the Kronecker product. In the case of the product code used in CORE, we will consider that the single parity check (SPC) with generator matrix $G_\mathbf{c} = [I_t, \mathbf{1}_t]$, i.e., a $(t+1, t)$ MDS erasure code over $\mathbb{F}_{2^q}$ is the **vertical** code. For an input $\mathbf{o} = (o_1, \dots, o_t)$, $o_i \in \mathbb{F}_q$, this code generates a systematic codeword $\mathbf{c} = (c_1, \dots, c_{t+1}) = (o_1, \dots, o_t, c_{t+1})$, where $c_{t+1} = \sum_{i=1}^{t} o_i$. Since $\mathbb{F}_{2^q}$ is a binary extension field the last symbol in the codeword corresponds to the exclusive-or (XOR) of the t original symbols. It can repair any single erasure in the codeword by xoring the remaining t symbols. The inner (**horizontal**) code $G_\mathbf{o}$ used in CORE is a MDS (n, k) erasure code. For the sake of simplicity, we will consider that it is a (n, k) Reed-Solomon code with generator matrix $G_\mathbf{o} = [I_k, H]$, where H is a $k \times m$ Vandermonde matrix (recall that $m = n - k$),

$$H = \begin{pmatrix} \alpha_1^0 & \dots & \alpha_1^{m-1} \\ \vdots & \ddots & \vdots \\ \alpha_k^0 & \dots & \alpha_k^{m-1} \end{pmatrix},$$

for any $\alpha_i \in \mathbb{F}_{2^q}$. Then, the CORE's product code is a linear code that cross-encodes t different data objects using a generator matrix $G = G_\mathbf{c} \otimes G_\mathbf{o}$. We will refer to such a code as a (n, k, t) CORE product code.

Lastly, it is worth noting that by varying the value of t, CORE allows for tuning the trade-off between good repairability (small t) and good storage overhead (large t). In our analytical study [6] as well as in our experiments we have chosen $t \approx k/2$, in order to strike a balance between the two criteria.

V. CORE's Algorithmic Aspects

One of the new aspects of CORE is its higher level of granularity, i.e., instead of working with individual independent objects, it works with a matrix of t objects. This higher granularity provides new opportunities (e.g., in a CORE scheme of (n, k, t) it is possible to repair an object that has more than $n-k$ failed blocks) and also poses new challenges (e.g., given a pattern of failures, is it possible to *recover* – i.e. repair all the failed blocks – the CORE matrix? or what is the best schedule for repairing a set of failures?).

In this section we look at these algorithmic problems and provide solutions for them. We adopt a divide-and-conquer approach to tackle these issues. Specifically, given a matrix representing the available and failed nodes (subsequently called the CORE matrix), we first split the failures into *independent clusters* (defined below). Other algorithms, i.e., recoverability-checking and repair scheduling, can be performed within each cluster. We discuss these next.

A. Identifying Independent Clusters

We define disjoint subsets of failed nodes that can be handled without interference as independent clusters.[2] Essentially, two different clusters must not share any common row or column containing failed nodes. Two important benefits of such clusters are (i) they allow parallel repairs, (ii) they may allow partial recovery when the full CORE matrix in not recoverable.

A naive way to create the clusters is as follows. Initially, each single failure is considered a cluster. Two clusters are then merged if there exists at least one common row or column on which both clusters have a failure. The process is continued until there are no mergeable clusters left. The number of clusters in a CORE matrix is between 0 and t (number of rows). To investigate the distribution of the number of clusters based on the number of failures, we ran our clustering algorithm on 10M randomly-generated CORE matrices for code parameters (14,12,5) and varied the number of random failures from 1 to 20. The results, depicted in Figure 3, show that after an initial increase, the

[2]In other parts of this paper, we also use the term computer/node cluster in the common sense of the word, which should not be confused with the failure clusters in the CORE matrix

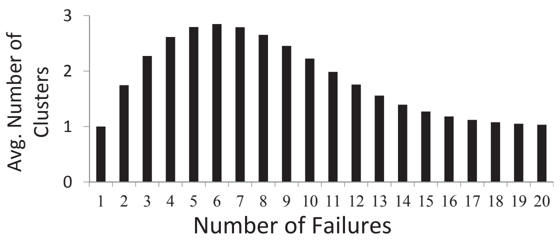


Figure 3: The average number of clusters versus the number of failures for CORE's code parameters (14,12,5)

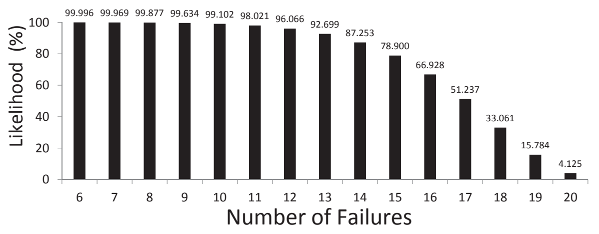


Figure 4: The recoverability likelihood of the scheme (14,12,5) based on the number of failures.

number of clusters begins to drop for failure numbers greater than 6.

B. Recoverability-Checking Algorithm

For coding schemes that work at the level of single objects, given the number of failures, one can directly infer whether an object is recoverable or not. In the case of CORE, however, this is more subtle. For instance, objects may still be recoverable even if there are more than $n - k$ failed blocks within a single CORE row. We first identify two bounds and then introduce an algorithm to determine an object's recoverability.

The (Ir)Recoverability Bounds. For a (n, k, t) code:

● the *lower* bound of *irrecoverability*, L, is:

$$2 \times (n - k + 1)$$

It occurs if two[3] rows are minimally irrecoverable (each has $n - k + 1$ failures) and the column indexes of their failures are identical (i.e., no vertical repair possible).

● the *upper* bound of *recoverability*, U, is:

$$t \times (n - k) + (2k - n) \times 1$$

This occurs when all rows are maximally recoverable (each has $n-k$ failures) and have identical failure column indexes (i.e., the remaining $k - (n - k) = 2k - n$ columns can each tolerate a single failure).

These two bounds define an interval. For any failure number outside of this interval, the ir/recoverability can be immediately decided. More precisely, if the number of failures is smaller than L then the pattern is recoverable – although, as we will see later, this is a very pessimistic bound – likewise, if the number is greater than U, then it is certainly not recoverable.

For all the values within the above interval (inclusive), the outcome depends on the distribution of the failures. We propose a recursive algorithm which is able to decide whether a given CORE matrix with a specific failure pattern is recoverable or not. At each step of the algorithm, all the repaired and repairable rows/columns are removed and the algorithm restarts with the reduced matrix as the new input. If it results in an empty matrix, then the patterns is recoverable, otherwise it is not.

[3]Any single-row failure pattern is always recoverable.

We implemented this algorithm and used it to carry out an analysis on the recoverability likelihood of different patterns. Figure 4, obtained from 10M random runs, shows the recoverability likelihood of the CORE matrix of size (14,12,5) for failure numbers between the *lower bound of irrecoverability* ($L = 6$) and the *upper bound of recoverability* ($U = 20$). It clearly illustrates the fact that CORE's lower bound of irrecoverability is too strict.

For a detailed and more rigorous study of fault-tolerance and recoverability, we refer the reader to our technical report [6].

C. Repair Scheduling Algorithms

Many different repair schedules may exist for a given fault pattern. Here, we first investigate two straw man approaches, namely *column-first* and *row-first*, then propose an algorithm called Recursively Generated Schedule (RGS). Analytical and experimental studies show that RGS outperforms the baseline approaches.

The *column-first* algorithm always gives higher priority to vertical repairs and applies horizontal repair when no further vertical repairs are possible. The *row-first* analogously prefers horizontal repairs. In both algorithms, while doing horizontal repairs, always the best candidate (the one with maximum number of failures but still repairable) is prioritized over the other ones.

Recursively Generated Schedule (RGS) algorithm:. This algorithm first identifies the critical set of failures (failures that decrease the minimum number of required vertical or horizontal repairs) and repairs them first, along the call chain of a recursive cost function c. All other repairs (non-critical ones) are then scheduled using c', a simple, non-recursive cost function.

In order to identify the critical failures, we define two variables, v and h, as follows:

$$v = \sum_{i=1}^{t} minV(Row_i) \qquad ; \qquad h = \sum_{j=1}^{k} minH(Col_j)$$

in which, $minV(Row_i)$ returns the minimum number of *vertical* repairs required by row Row_i, and $minH(Col_j)$

returns the minimum number of *horizontal* repairs required by column Col_j, more precisely:

$$minV(Row_i) = \begin{cases} 0 & \text{if } |X| \leq (n-k) \\ |X| - (n-k) & \text{otherwise} \end{cases}$$

$$minH(Col_j) = \begin{cases} 0 & \text{if } |X| \leq 1 \\ |X| - 1 & \text{otherwise} \end{cases}$$

The most important element of RGS is the recursive cost function $c(h, v)$ defined as:

$$c(h,v) = \begin{cases} c(h, dec(v)) + t & \text{if } v > 0 \\ \\ c(dec(h), v) + k & \text{if } v = 0 \\ & \text{or } dec(v) \text{ is not applicable} \end{cases}$$

in which $dec(v)$ and $dec(h)$ reflect the decreases in the values of v and h after a single repair is performed.

The cost function c decreases the values of first v and then h by at least one unit at each recursion step until we reach $c(0, 0)$, which is the *base case*[4]. The notable property of the base case is that any remaining repair can be done either vertically or horizontally. In other words, there is at most one failure per column, and at most $n - k$ failures per row. Therefore, all remaining repair decisions can be safely made using the static cost function c' defined below:

$$c'(r) = \begin{cases} k & \text{if repaired horizontally} \\ r \times t & \text{if repaired vertically} \end{cases}$$

in which r denotes the number of remaining repairs for a given row.

To demonstrate the differences between the repair schedules generated by the above three algorithms, we use two failure pattern examples in the CORE matrix of size (14,12,5): a 3-failure *step*-shaped pattern and a 5-failure *plus*-shaped one. These examples are shown in Table I.

$$\begin{pmatrix} \dots & 0 & 0 & \dots \\ \dots & 0 & 0 & \dots \\ \dots & \mathbf{x} & 0 & \dots \\ \dots & \mathbf{x} & \mathbf{x} & \dots \\ \dots & 0 & 0 & \dots \end{pmatrix} \qquad \begin{pmatrix} \dots & 0 & 0 & 0 & \dots \\ \dots & 0 & \mathbf{x} & 0 & \dots \\ \dots & \mathbf{x} & \mathbf{x} & \mathbf{x} & \dots \\ \dots & 0 & \mathbf{x} & 0 & \dots \\ \dots & 0 & 0 & 0 & \dots \end{pmatrix}$$

Table I: The step-shaped and the plus-shaped failure pattern examples representing two classes of failure patterns.

It should be noted that since swapping any two rows or any two columns in the CORE matrix results in an equivalent failure matrix, each of these patterns represents a class of failure patterns and not singular instances. Table II presents the schedules generated by each algorithm for each failure pattern along with its calculated cost in terms of repair traffic. The corresponding experimental results are reported in Section VII.

Finally, we generalized our analytical study of the above three algorithms to include failure patterns of size 1 to 20. The results for 10,000 randomly-generated recoverable failure patterns are depicted in Figure 5. Four conclusions

[4]If the failure pattern is recoverable, the base case will always be reached.

		Row-First	**Column-First**	**RGS**
Step	*Schedule*	R_3, R_2	C_1, R_2, C_0	$c(1,0) \xrightarrow{R_3} c(0,0) \rightarrow C_1$
	Cost	$2k = 24$	$2t + k = 22$	$k + t = 17$
Plus	*Schedule*	R_1, R_3, C_0, R_2	C_0, C_2, R_1, R_2, C_1	$c(2,1) \xrightarrow{C_0} c(2,0) \xrightarrow{R_2} c(1,0) \xrightarrow{R_1} c(0,0) \rightarrow C_1$
	Cost	$3k + t = 41$	$3t + 2k = 39$	$2t + 2k = 34$

Table II: The analytical cost (number of blocks read) of repairing the Step and Plus failure patterns using Row-First, Column-First, and RGS algorithms where $k = 12$ and $t = 5$.

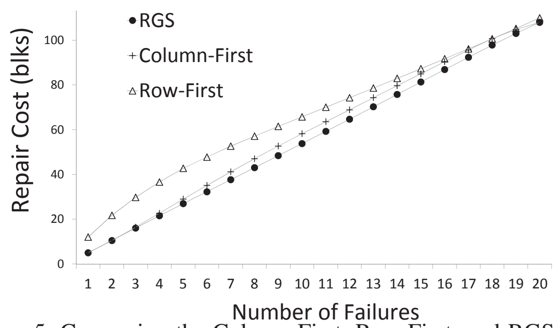


Figure 5: Comparing the Column-First, Row-First, and RGS algorithms w.r.t number of blocks required to carry out the repair on the scheme (14,12,5).

can be drawn from this figure: (i) RGS and column-first perform better than row-first and this is especially noticeable when the number of failures is very small (which is, in essence, the MDS code vs. CORE comparison); (ii) as the number of failures and consequently the number of choices to make increases, the benefits of RGS over column-first become more pronounced; (iii) for the large failure numbers, distinct schedule possibilities are limited, and all the algorithms perform similarly; and finally (iv) a more general conclusion is that if one wishes to avoid the relatively complex scheduling algorithms, then the naive column-first approach nevertheless delivers significant benefits w.r.to the row-first (which is roughly like for MDS codes), highlighting the immediate benefits of CORE's product code.

VI. Implementation

To implement the CORE primitive, we used HDFS-RAID [2], an open-source module inspired by DiskReduce [14], and developed at Facebook. It wraps around Apache Hadoop's distributed file system (HDFS) and provides HDFS with basic erasure coding capabilities (encoding and decoding). Below, we first introduce HDFS-RAID, then explain two optimizations that we did on HDFS-RAID to improve its performance, and finally give an overview of our implementation of CORE.

A. HDFS-RAID

HDFS-RAID embeds the Apache HDFS inside an erasure code-supporting wrapper file system named Distributed Raid File System (DRFS). DRFS supports both Reed-Solomon coding as well as simple XOR parity files. These two coding alternatives are orthogonal and used separately based on user preference. Furthermore, both provide two basic features: encoding (a.k.a RAIDing) data blocks and repairing the corrupt/missing blocks.

The two main components of HDFS-RAID are RaidNode and BlockFixer. RaidNode is a daemon responsible for the creation (only once, following the initial file write) and maintenance (re-creating periodically or on demand the corrupt/missing parities and purging "orphan" ones) of parity files for all data files. Since the default block policy of HDFS is not aware of the dependency relation between the data and parity blocks of a given file, HDFS-RAID manages the placement of parity blocks to avoid co-location of data blocks and parity blocks. The BlockFixer component reconstructs missing or corrupt blocks by retrieving the necessary blocks, encoding/decoding them, and sending the reconstructed blocks to new hosts.

B. HDFS-RAID Optimizations

In our experiments with HDFS-RAID, we noticed two common performance inefficiencies, and optimized them:

Opt1: The HDFS-RAID implementation uses the *generator polynomial* (and not the more well-known *generator matrix* [32]) representation of Reed-Solomon codes. In this representation, typically and as is in the HDFS-RAID implementation, always *all* the remaining blocks of a given row (which can be more than k) are fetched and used to repair the missing ones. Generally, this use of extra blocks results in faster decoding, since there will be fewer equations to solve. However, for cases in which network is a bottleneck, this trade-off (fetching extra blocks versus faster decoding) does not pay off. Our optimized version retrieves exactly *k* blocks and "pretends" that all other $n - k$ blocks are missing. As confirmed by our experimental results, the bandwidth-scarce clusters can greatly benefit from this optimization.

Opt2: The HDFS-RAID implementation implicitly assumes that there is only a single failure per row (stripe). In case there are more failures, they are discovered only when the read access attempts fail. These newly-detected failed blocks are then added to the list of failed blocks, and the repair process starts again. Our optimized implementation checks for multiple failures beforehand, and repairs them simultaneously, amortizing the repair costs.

C. CORE Implementation

The CORE storage primitive has been organically integrated with HDFS-RAID by extending its two main functionalities as described below.

RAIDing: The CORE implementation allows vertical coding across files in a given directory. The cross-object size parameter (t) can be configured similar to the row (stripe) size parameter of HDFS-RAID. The vertical encoding is reused in the full matrix RAIDing (first row-by-row, then column-by-column, for both data and parity blocks).

Repair: An additional vertical repair option is introduced. The 2-dimensional repair feature implements all the algorithms discussed in Section V: (i) failure detection and failure matrix population, (ii) failure clustering, (iii) recoverability-checking, and (iv) repair scheduling.

The correctness of our implementation was verified through multiple test cases in which the MD5 hash values of the repaired files were compared against those of the original files. Moreover, since all changes have been made within the RAID subdirectory of the HDFS's code, replacing the corresponding Java library is sufficient to upgrade HDFS-RAID to CORE. The source codes, binary distribution, and documentations of our implementation are available at http://sands.sce.ntu.edu.sg/StorageCORE.

VII. Experiments

We benchmarked the implementation with experiments run on two different HDFS clusters of 20 nodes each:

• **Network-Critical** cluster: A university cluster which has one powerful PC (4×3.2GHz Xeon Processors with 4GB of RAM) hosting the NameNode/RaidNode and 19 HP t5745 ThinClients acting as DataNodes. The average bandwidth of this cluster is 12MB/s.

• **Computation-Critical** cluster: An Amazon EC2 cluster of 20 homogeneous nodes of type *m1.small* (approximately, 1.2 GHz 2007 Xeon Processor with 1.8GB of RAM). In this cluster one node is hosting the NameNode/RaidNode and the rest are used as DataNodes. The maximum bandwidth between EC2 *m1.small* instances is 250MB/s.

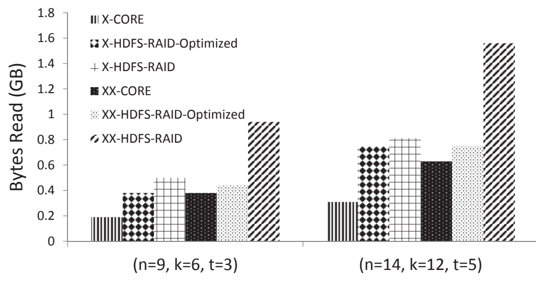
The block size (q) used was 64MB. Files were added to HDFS and encoded horizontally first, and then the vertical parity was computed.

We ran two sets of experiments, one set to compare the performance of CORE with that of HDFS-RAID, and another set to study the repair scheduling algorithms. In both sets, we primarily use the **completion time** of the repair process as the main comparison measure. However, we also measured the amount of **transferred data** in each experiment (as repair traffic). The data transfer numbers serve two purposes: (i) to verify the correctness of our implementation –they must match the analytical numbers– and (ii) to use as a reference point in analyzing the completion time numbers – since the amount of transferred data is independent of the type of cluster used.

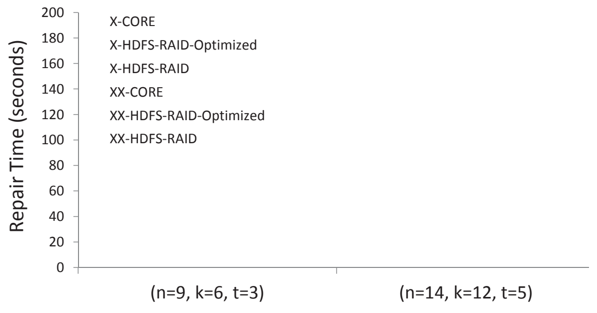
Finally, in all experiments the reported numbers are the average of 10 runs. Since the variations were small (up to few percents), they are omitted from the graphs.

A. CORE vs. HDFS-RAID

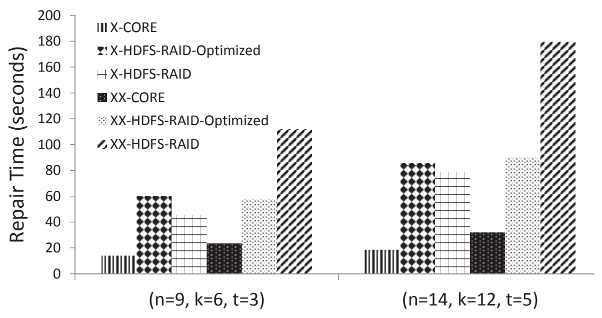
In these experiments, we compared three methods (namely, HDFS-RAID, HDFS-RAID-Optimized and CORE)



(a) Transferred data



(b) Time (network-critical cluster)



(c) Time (computation-critical cluster)

Figure 6: Comparing the repair performance of HDFS-RAID, HDFS-RAID-Optimized, and CORE

using two different sets of coding parameters: (9,6,3) and (14,12,5), inspired respectively by the code length and storage overheads of Google's GFS and Microsoft Azure. In these schemes, the overhead of CORE's extra parities are $1/3 = 33\%$ and $1/5 = 20\%$ accordingly.

In each case two different failure patterns were enforced: a one-failure pattern represented by **X** and a two-failures pattern represented by **XX**. For the two-failures pattern, both are set to happen in the same object (i.e., on the row). The reason for this setting is two-fold: (i) it favors the HDFS-RAID since at almost the same cost it can repair two failures instead of one; (ii) if two failures happen on different rows, the experiment will be, in effect, a variation of the one-failure pattern.

From the results shown in Figure 6, we can draw several conclusions:

• For single failure, the overhead of CORE is less than 50% of HDFS-RAID. This is quite significant, since in real-world clusters, e.g., in the Facebook cluster [33], single failures (per stripe) are by far the most common type of failures. This improvement results from two inherent advantages of CORE: (i) single failure can be repaired vertically, using far fewer blocks, and (ii) it uses a much cheaper XOR operation instead of expensive decoding/re-encoding (this is particularly significant in the computation-critical cluster).

• The impact of our first HDFS-RAID optimization (Opt1 in Section VI-B) can be seen in the results (the difference between the 2nd and the 3rd chart bars). As explained before, this optimization is targeted *specifically* for the clusters in which the network is a scarce resource (part *b* in Figure 6). The improvements are particularly pronounced in cases where the number of avoided block retrievals are higher (e.g., one failure in the scheme (9,6,3)).

• The gains from our second HDFS-RAID optimization (Op2 in Section VI-B) are also noticeable (the 5th and the 6th chart bars in all setups).

• Growth in the CORE matrix size, from (9,6,3) to (14,12,5), results in even higher gains, especially in clusters where computation power is scarce.

B. *Repair Scheduling Algorithms*

In this set of experiments, the three repair scheduling algorithms of Section V-C were compared using the *Step* and *Plus* failure patterns. HDFS-RAID has neither a notion of repair scheduling – it treats objects independently – nor can it fully recover from the Plus failure pattern, so it was not considered in the following experiments.

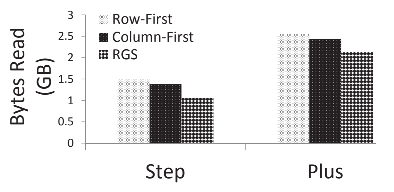
These experiments were run for CORE matrix of size (14,12,5). The results are shown in Figure 7 and as expected, the data part of this figure (part *a*) mirrors the analytical results presented in Table II. Moreover, the completion time numbers (parts *b* and *c*) are also, to large extent, in-line with the data results. The only two discrepancies are explained below:

• Completion time of the Column-First algorithm on the Plus pattern in the network-critical cluster (part *b*) is longer than expected. This is caused by the last repair which uses two other freshly-repaired blocks. Accessing those blocks is delayed until NameNode's heartbeat-driven mapping tables are updated.

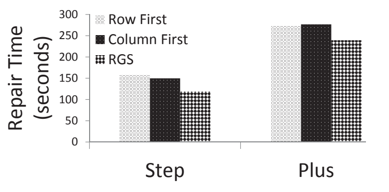
• Completion time of the RGS algorithm in the computation-critical cluster (part *c*) is only slightly better than that of Column-First, despite applying one vertical repair less (see Table II for the schedules). This is due to the fact that for these patterns the RGS and Column-First apply the same number of horizontal repairs and these are the main driving factor of the cost in the computation-critical cluster.

VIII. Conclusions and Future Work

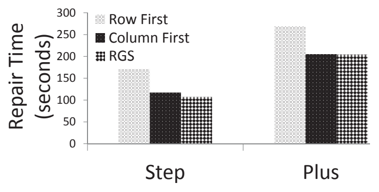
In this paper we demonstrated that some simple and standard techniques (and thus easy to implement and organically



(a) Transferred data



(b) Time (network-critical cluster)



(c) Time (computation-critical cluster)

Figure 7: Performances of the repair scheduling algorithms on two different failure patterns.

integrate) can provide significant data repair and access boost in erasure coded distributed storage systems. Specifically, we studied our approach of introducing cross-object coding on top of normal erasure coding. The ideas were implemented and integrated with HDFS-RAID (available at [7]), and benchmarked over a proprietary cluster and EC2. Experiments with the implementation (as well as accompanying analytical studies [6] comparing the approach with not only MDS codes but also with the very recently proposed Local Reconstruction Codes used in Azure) demonstrate the superior performance of CORE over state-of-the-art techniques for data reads and repairs. While naive solutions can be readily used, in future we will like to explore the CORE code properties to achieve better performance also during data insertion/updates. The current evaluations are static, based on snapshots of the system state. We speculate that CORE's better repair properties will yield a system in a better state over time. We will thus carry out trace driven experiments to study the system's dynamics better.

References

[1] P. Elias, "Error-Free Coding," *Transactions on Information Theory*, vol. 4, no. 14, 1954.
[2] HDFS-RAID, http://wiki.apache.org/hadoop/HDFS-RAID.
[3] C. Huang et al., "Erasure Coding in Windows Azure Storage," in *USENIX ATC*, 2012.
[4] A. Datta et al., "Redundantly Grouped Cross-object Coding for Repairable Storage," in *Proc. APSys*, 2012.
[5] P. Gopalan et al., "On the locality of codeword symbols," *Information Theory, IEEE Transactions on*, vol. 58, no. 11, pp. 6925–6934, 2012.
[6] K. S. Esmaili et al., "The CORE Storage Primitive: Cross-Object Redundancy for Efficient Data Repair and Access in Erasure Coded Storage," *CoRR*, vol. abs/1302.5192, 2013.
[7] CORE, http://sands.sce.ntu.edu.sg/StorageCORE.
[8] H. Weatherspoon et al., "Erasure Coding vs. Replication: A Auantitative Comparison," in *Proc. IPTPS*, 2002.
[9] J. Kubiatowicz et al., "OceanStore: An Architecture for Global-Scale Persistent Storage," in *Proc. ASPLOS*, 2000.
[10] R. Bhagwan et al., "Total Recall: System Support for Automated Availability Management," in *NSDI*, 2004.
[11] S. Plank, "The RAID-6 Liber8Tion Code," *Intl. Journal of High Performance Computing Applications*, vol. 23, no. 3, 2009.
[12] A. Patterson et al., "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *SIGMOD Records*, vol. 17, no. 3, 1988.
[13] O. Khan et al., "Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads," in *USENIX FAST*, 2012.
[14] B. Fan et al., "DiskReduce: Replication as a Prelude to Erasure Coding in Data-Intensive Scalable Computing," CMU, Tech. Rep. CMU-PDL-11-112, 2011.
[15] B. Calder et al., "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," in *ACM SOSP*, 2011.
[16] A. Thusoo et al., "Data Warehousing and Analytics Infrastructure at Facebook," in *ACM SIGMOD*, 2010.
[17] C. Huang et al., "Pyramid Codes: Flexible Schemes to Trade Space for Access Efficiency in Reliable Data Storage Systems," in *IEEE NCA*, 2007.
[18] F. Oggier et al., "Coding Techniques for Repairability in Networked Distributed Storage Systems," *FnT in Communications and Information Theory*, vol. 9, no. 4, 2013.
[19] A. Dimakis et al., "A Survey on Network Codes for Distributed Storage," *The Proc. of IEEE*, vol. 99, 2011.
[20] A. Duminuco et al., "Hierarchical Codes: How to Make Erasure Codes Attractive for Peer-to-Peer Storage Systems," in *Proc. P2P*, 2008.
[21] M. Li et al., "GRID Codes: Strip-Based Erasure Codes with High Fault Tolerance for Storage Systems," *ACM Trans. on Storage*, vol. 4, 2009.
[22] A. Kermarrec et al., "Repairing Multiple Failures with Coordinated and Adaptive Regenerating Codes," in *Proc. NetCod*, 2011.
[23] K. W. Shum, "Cooperative Regenerating Codes for Distributed Storage Systems," in *Proc. ICC*, 2011.
[24] F. Oggier et al., "Self-Repairing Homomorphic Codes for Distributed Storage Systems," in *Proc. INFOCOM*, 2011.
[25] F. Oggier et al., "Self-Repairing Codes for Distributed Storage - A Projective Geometric Construction," in *Proc. ITW*, 2011.
[26] D. Papailiopoulos et al., "Locally Repairable Codes," in *Proc. ISIT*, 2012.
[27] M. S. et al., "Xoring elephants: Novel erasure codes for big data," *Proceedings of the VLDB'13 (To appear)*, 2013.
[28] L. Pamies-Juarez et al., "Data Insertion & Archiving in Erasure-coding Based Large-scale Storage Systems," in *Proc. ICDCIT*, 2013.
[29] L. Pamies-Juarez et al., "RapidRAID: Pipelined Erasure Codes for Fast Data Archival in Distributed Storage Systems," in *Proc. INFOCOM*, 2013.
[30] Y. Hu, "NCFS: On the Practicality and Extensibility of a Network-Coding-Based Distributed File System," in *Proc. NetCod*, 2011.
[31] R. Li et al., "CORE: Augmenting Regenerating-coding-based Recovery for Single and Concurrent Failures in Distributed Storage Systems," in *Proceedings of IEEE MSST'13*, 2013.
[32] J. I. Hall, *Notes on coding theory*. Citeseer, 2003.
[33] K. V. Rashmi and others., "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster," in *Proceedings of USENIX HotStorage'13*, 2013.