2015 44th International Conference on Parallel Processing

EC-FRM: An Erasure Coding Framework to Speed up Reads for Erasure Coded Cloud Storage Systems

Yingxun Fu, Jiwu Shu*, and Zhirong Shen
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University
Beijing, China, 10084
Corresponding Author: shujw@tsinghua.edu.cn
mooncape1986@126.com, zhirong.shen2601@gmail.com

***Abstract*—With the reliability requirements increasingly important, erasure codes have been widely used in today's cloud storage systems because they achieve both high reliability and low storage overhead. However, the performance for most existing erasure codes can be further improved on both normal reads to user's data without device failures and degraded reads under device failures, which are crucial in cloud storage systems. In this paper, we propose an erasure coding framework named EC-FRM to integrate existing codes in order to improve the read performance. The constructed code over EC-FRM named EC-FRM-Code, which keeps most of wonderful properties of the integrated code and achieves good performance on both normal reads and degraded reads. We transform Reed-Solomon code and LRC code to EC-FRM-RS and EC-FRM-LRC respectively, and then conduct a series of experiments to evaluate their read performance. The results show that EC-FRM-RS code gains 19.2% to 33.9% higher normal read speed and 9.1% to 9.9% higher degraded read speed than standard Reed-Solomon code, while EC-FRM-LRC code owns 23.5% to 46.9% higher normal read speed and 3.3% to 12.8% higher degraded read speed than standard LRC code.**

***Keywords*-Erasure Code; Cloud Storage System; Normal Read; Degraded Read;**

I. Introduction

With the data size rapidly increasing, erasure codes have received more and more attentions in today's cloud storage systems because they achieve both low storage overhead and high reliability [1][2]. Cloud storage system using erasure code to assure the reliability forms erasure coded cloud storage system, which transfers the requirements of traditional erasure codes due to the fact that they usually adopt append-only write workloads and large block size.

In erasure coded cloud storage systems, writes are usually accumulated in the end of the files and buffered in memory or devices until a block is fully written and then the blocks is erasure coded (i.e., full stripe writes) [3]. Since existing erasure codes usually perform similarly on full stripe writes, the performance on writes is not significant. Since the GF-Complete [4] library significantly improves the Galois Field's arithmetic efficiency, the encoding and decoding computation performance between various codes are no much different compared with the I/O efficiency [3]. On the other hand, reads turn into very crucial performance metric and receive a lot of attentions because they occur very frequently [5].

However, most previous works put their eyes on how to improve the degraded read performance (i.e., reads under disk failures), but ignore a fact that normal reads (i.e., reads when all disks are available) usually appear more frequently than degraded reads. By assuming the contiguous symbols are stored on different disks [3] and the bandwidth is sufficient (e.g., in inner-enterprise cloud storage), the read performance in most horizontal codes (the codes store all parities in specialized disks, such as Reed-Solomon code [6] and LRC code [5]) can be further improved. For example, a (6,2,2) LRC code has 6 data disks and 4 parity disks, when it reads to 7 data elements, there exists at least one disk needing to afford 2 elements and thus degrades the performance, because the read speed is usually limited by the slowest disk to respond the reading data. Yet, if there exist more than 7 disks can help this read operation (there exist up to $6+4=10$ potentially contributed disks), the performance will be further improved. Though some methods like rotating mappings from logic disks to physical disks are able to alleviate this problem in some level, the read performance under these methods still has some potential improvements. On the other hand, though many vertical codes (the codes distributed parities among all disks) like WEAVER code [7] and X-Code [8] provide good performance on normal reads, they usually cannot achieve high fault tolerance and low storage overhead simultaneously, and usually cannot apply to arbitrary number of disks. Due to these shortcomings, traditional vertical codes are rarely used in today's erasure coded cloud storage systems [3][5].

In this paper, we propose a new erasure coding framework named EC-FRM to integrate existing codes, in order to improve the read performance. EC-FRM is constructed by a series of mathematical equations, which can integrate some original erasure codes and redeploy their data and parities. The integrated code is named EC-FRM-Code, which not only keeps most merits of the original code, but also achieves

0190-3918/15 $31.00 © 2015 IEEE
DOI 10.1109/ICPP.2015.57

good performance on both normal reads and degraded reads. The contributions of this paper can be summarized as follows:

- We propose a novel erasure coding framework termed EC-FRM, which can integrate existing codes like Reed-Solomon code and LRC code while keeping most merits of these codes. E.g., EC-FRM-RS code (EC-FRM over Reed-Solomon code) provides MDS properties and can tolerate arbitrary number of disk failures, and EC-FRM-LRC code (EC-FRM over LRC code) significantly reduce the I/O accesses on degraded reads.
- We implement two concrete EC-FRM-Code based on Jerasure 1.2 [21], and evaluate their read performance in a real system. The experiment results illustrate that EC-FRM-Code performs well on both normal reads and degraded reads. Specifically, EC-FRM-RS gains 19.2% to 33.9% higher normal read speed and 9.1% to 9.9% higher degraded read speed than standard Reed-Solomon code, while EC-FRM-LRC code owns 23.5% to 46.9% higher normal read speed and 3.3% to 12.8% higher degraded read speed than standard LRC code, respectively.

The rest of this paper continues as follows: the next section states the background and related work. Section III discusses the problem in existing erasure codes and gives our motivations. Section IV represents the detailed designs of EC-FRM. Section V analyzes the properties. We focus on experimental evaluations in Section VI. Finally, we conclude the paper in the last section.

II. Background and Related Work

A. Terms and Notations

Element: Element is the fundamental unit for erasure codes, which is a chunk of data or parity information. There mainly exist two kinds of elements: data elements which contain the original data information, and parity elements that keep the redundant information. In Figure 1, $d_{0,0}$ is a data element, while $p_{0,0}$ is a parity element.

Stripe: Stripe is the maximal set of elements that are dependent of each other in terms of layout and encoding relationships [10]. Figure 1 and 2 illustrate a stripe of Reed-Solomon code [6] and LRC code [5], respectively.

Row: Row is an maximum set of elements that belong to the same row of stripe. In Figure 1, all elements constitute a specific row, because a stripe of Reed-Solomon code comprises only one row.

Rotated Stripes: In order to balance the I/O loads on each disk, some storage systems rotate the mapping from logic disks to physical disks stripe by stripe. In this paper, we call these stripes as rotated stripes.

B. The Erasure Codes

Many erasure codes have been proposed in recent years. Based on the parity distributions, erasure codes can be simply classified into horizontal codes and vertical codes.

Horizontal Codes: Horizontal codes store their parities in specialized parity disks, which can be further categorized into $GF(2^n)$ based codes and XOR-Based codes according to different encoding rules. The $GF(2^n)$ based erasure codes usually can tolerate arbitrary disk failures, including Reed-Solomon code [6], Rotated Reed-Solomon code [3], LRC code [5], SD code [18], and STAIR code [19], of which Reed-Solomon code is an MDS [13] code and thus provides the optimal storage efficiency. LRC code sacrifices some storage rate to minimize the I/O accesses on degraded reads. Rotated Reed-Solomon code utilizes the overlapping elements to reduce the I/O accesses on degraded reads. SD code and STAIR code are new kinds of erasure codes that focus on both disk failures and sector failures.

There also exist some XOR-Based horizontal codes, such as Cauchy Reed-Solomon code [16], RDP code [14], Liberation code [15] and STAR code [20]. Cauchy Reed-Solomon code is an improved Reed-Solomon code and can tolerate arbitrary disk failures. RDP code and Liberation code are classic RAID-6 codes and can tolerance just up to 2 disk failures.

Vertical Codes: Vertical codes distribute parities among all disks, including X-Code [8], C-Code [24], D-Code [25], HDP code [17], HV-Code [26], and WEAVER code [7], which are usually constructed by only XOR operations and provide good encoding/decoding complexity and good load balancing. But unfortunately, they cannot achieve both high fault tolerance and low storage overheads simultaneously. For example, X-Code and P-Code are RAID-6 codes and just tolerate up to 2 disk failures, while WEAVER code always provides no more than 50% storage usage ratio. Moreover, vertical codes usually cannot apply to arbitrary number of disks.

C. Erasure Codes for Cloud Storage Systems

Though such erasure codes have been proposed in literature, there are just a few implementations appeared in modern cloud storage systems. In this subsection, we discuss two representative and widely used erasure codes.

Reed-Solomon Code for Google: Reed-Solomon code is a classic erasure code and can tolerate arbitrary number of disk failures. The encoding of Reed-Solomon code is based on both addition operations and multiply operations, where the addition is based on XOR operations and the multiply is based on a finite field $GF(2^w)$ [13]. A specifical Reed-Solomon code can be represented as a two-tuple (k, m), where k and m represent the number of data disks and parity disks, respectively.

Figure 1 shows an example of (6,3) Reed-Solomon code. As it shows, a stripe of Reed-Solomon code just contains only one row, because the layout and the encoding relationship of arbitrary two elements in different rows are independent of each other. In order to save storage space,

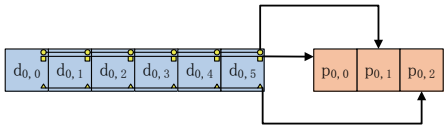


Figure 1: An example of (6,3) Reed-Solomon code.

Google uses the (6,3) Reed-Solomon code to substitute triple-replications [1], while Facebook implements the Reed-Solomon code in its local storage systems as well [2].

LRC Code for Azure: LRC code is another typical erasure code for reducing the I/O cost on degraded reads. There are two types of parities in LRC code: local parity, which is calculated by a part of data elements in each row, and global parity which is computed by all data elements of each row. A specifical LRC code can be represented by a three-tuple (k, l, m), where k indicates the number of data disks, l denotes the number of local parity disks, and m represents the number of global parity disks.

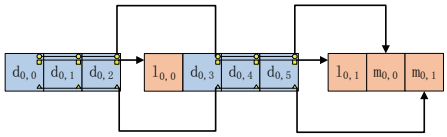


Figure 2: An example of (6,2,2) LRC code.

Figure 2 shows an example of (6,2,2) LRC code. As the figure shows, the local parity elements ($l_{0,0}$ and $l_{0,1}$) are calculated by three data elements, while the global parity elements ($m_{0,0}$ and $m_{0,1}$) are computed by all data elements.

D. Performance Metrics

With the rapid growth of data scale, disk failures are normal things in today's storage systems [1]. Among the diverse failure patterns, previous study [11] has revealed that most of recoveries (up to 99.75%) are triggered by single disk failures. Based on this observation, recently studies [3][27][28] point out that there are two crucial performance metrics emerged in cloud storage system: recovery from single failures and **degraded reads** to respond users' read requests. Huang et al. [5] further indicate that the performance on degraded reads is more significant than that on single failure recoveries, because more than 90% of data center failures are triggered by the system upgrading with no data lost [1]. On the other hand, it is easy to deduce that the performance on reads without disk failures is another very important metric, because it is very frequently appeared in cloud storage systems [12][2]. In order to distinguish the above two types of reads, we denote the reads without disk failures as **normal reads**.

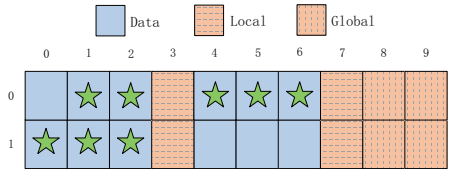
On the other hand, most cloud storage systems adopt append-only write strategy causing that the performance on writes are not as important as that on reads [3]. Moreover, since GF-Complete [4] library significant improves the Galois Field's arithmetic efficiency, the encoding and decoding performance between various codes are not much different compared to the I/O efficiency of storage devices and thus isn't so crucial.

III. Problem Statements and Motivations

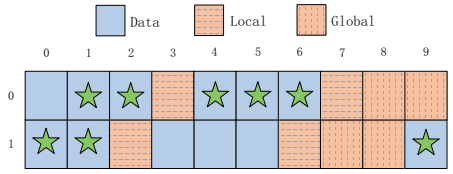
Different cloud storage systems usually have different network bandwidth[29][30]. In this paper, we focus on the kind of cloud storage systems with sufficient bandwidth (E.g., inner-enterprise cloud storage systems). We now state the problems in existing erasure codes and give our motivations.

A. Problem Statements

We first give an assumption that the contiguous elements user requests are stored on different disks in the storage system in order to take maximum advantage of parallel I/O (the same assumption is also applied in [3] and be widely used in cloud storage systems). Based on this assumption, horizontal codes usually cannot provide satisfied normal read performance, because 1) the parity disks don't contribute to normal reads 2) the read speed is restricted by the access time on the slowest disk, which is usually the most loaded disk. Now, we state this problem through a simple example.



(a) In conventional stripes.



(b) In rotated stripes

Figure 3: An example of 8-elements read operation in (6,2,2) LRC code.

Figure 3 shows two examples of read to 8-elements in (6,2,2) LRC code with standard stripes and rotated stripes, respectively. As Figure 3(a) shows, in order to read the required elements, disk 1 and disk 2 need to access two elements that form the bottleneck, because the other disks just need to access only one element. However, if all 10 disks can contribute to this read operation, the most loaded disks just need to afford one element's access, thus improving the read performance. The rotated stripes also cannot solve this problem, because the parity elements in rotated stripes laid on the same row with data elements and thus affect

the number of disks contributed to reads (As Figure 3(b) illustrates). In addition, degraded reads suffer from the similar problem.

One may argue that if the number of elements user requests is small (less than 6 in the case of Figure 3(a)), most of horizontal codes don't suffer from the read problems referred above. We follow this argument as well. However, previous study in [3] mentions that user requests more than 6 data elements is frequently appeared in real cloud storage systems, because the size of some common files (like MP3 files) is usually from a few megabytes to dozens of megabytes and the size of each elements in some storage systems is usually several megabytes (E.g., 1MB). Therefore, this observation inspired us that to improve the performance on several elements' reads is meaningful.

Moreover, though some traditional vertical codes such as X-Code and WEAVER code perform well on normal reads, they usually cannot simultaneously provide both high fault tolerance and low storage overhead, and usually cannot apply to arbitrary number of disks. Because of these shortcoming, vertical codes are rarely used in real cloud storage systems [3].

B. Motivations

Though many erasure codes have been proposed in literature, there only exist a few implementations in cloud storage systems, of which Reed-Solomon code has been widely used in Google and Facebook [23][2][1] and LRC code has been used in Windows Azure [5]. Therefore, the designed framework should keep the merits of integrated codes besides improving read performance. Based on this reason and the problems in existing codes discussed above, we have the following motivations.

Minimize I/O Accesses on The Most Loaded Disk: As referred above, the read performance is mainly restricted by the I/O accesses on the slowest disk. Since the most loaded disk is more likely to be also the slowest disk when the block size is large (such as disk 1 in Figure 3(a) and Figure 3(b)), our first motivation is to generate erasure coding schemes to minimize the I/O accesses on the most loaded disk.

Design Frameworks to Integrate Existing Codes While Keeping Their Merits: Existing codes such as Reed-Solomon code and LRC code have some absorbing properties, which attract the designers to choose them in the real cloud storage system implementations. Therefore, the second motivation of us is to design a framework to integrate popular codes in order to generate new codes that maintain the original codes' merits.

Adapt for Arbitrary Number of Disks: Existing vertical codes usually cannot apply to arbitrary number of disks, which restrict their usage in some storage applications [15]. Therefore, the designed framework needs to satisfy the condition that, if a code before transformation can apply to arbitrary number of disks, the new code after transformation must be able to apply to arbitrary number of disks as well.

IV. EC-FRM Description

A. Definitions

Candidate Code: Since EC-FRM needs to integrate other erasure codes to construct new coding schemes, we define candidate code as the code which can be integrated to EC-FRM. Candidate codes should satisfy the condition that each stripe contains only one row. For example, the widely used codes, Reed-Solomon code and LRC code, are candidate codes.

EC-FRM-Code: We use EC-FRM-Code to denote the coding scheme constructed by EC-FRM. Specifically, we use EC-FRM-RS code to denote the EC-FRM constructed upon Reed-Solomon code, and use EC-FRM-LRC code to indicate the EC-FRM constructed upon LRC code. For each EC-FRM-Code, we use the same parameters as its candidate code to describe it. E.g., if an EC-FRM-Code construct upon (6,2,2) LRC code, we label the constructed erasure coding scheme as (6,2,2) EC-FRM-LRC.

Group: In EC-FRM-Code, group is the maximum set of data elements and parity elements that are dependent of each other in encoding relationships. For each group, all parity elements are calculated by this group's data elements.

B. Layout and Construction Rules

We simply denote each candidate code as a two-tuple (n,k), where n indicates the total number of elements in each stripe (row) and k presents the number of data elements in each stripe (row), in order to describe EC-FRM's detailed designs in mathematics. For example, a (6,2,2) LRC code can be denoted as a (10,6) candidate code. We now discuss how to construct a generic EC-FRM-Code by its candidate code.

A stripe of any EC-FRM-Code have $\frac{n}{gcd(n,k)}$ rows and n columns, where the mathematics notation '$gcd(n,k)$' means the greatest common divisor between n and k. In order to clarify the layout of the EC-FRM-Code, we denote the ith data element of the jth column as $d_{i,j}$, denote the ith parity element of the jth column as $p_{i,j}$, and define the parameter r as:

$$r = gcd(n,k)$$

The data elements of any EC-FRM-Code have been laid in the first $\frac{k}{r}$ rows, while the parity elements are deployed in other rows. All elements in EC-FRM-Code can be classified as $\frac{n}{r}$ groups identified by G_i ($0 \leqslant i \leqslant \frac{n}{r}-1$). We label all data elements of G_i as D_i, denote all parity elements of G_i as P_i, and use $P_{i,j}$ to represent the intersection of P_i and the $(j+\frac{k}{r})th$ ($0 \leqslant j \leqslant \frac{n-k}{r}-1$) row. For example, as shown in Figure 4, if G_2 = $\{d_{1,2}, d_{1,3}, d_{1,4}, d_{1,5}, d_{1,6}, d_{1,7}, p_{3,8}, p_{3,9}, p_{4,0}, p_{4,1}\}$ and $\frac{k}{r} = 3$, we can deduce that D_2

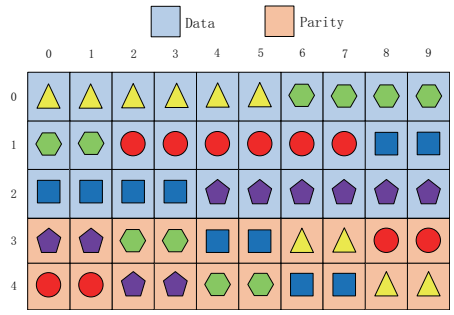


Figure 4: The layout of any (6,4) EC-FRM-Code.

= $\{d_{1,2}, d_{1,3}, d_{1,4}, d_{1,5}, d_{1,6}, d_{1,7}\}$, P_2 = $\{p_{3,8}, p_{3,9}, p_{4,0}, p_{4,1}\}$, $P_{2,0}$ = $\{p_{3,8}, p_{3,9}\}$, and $P_{2,1}$ = $\{p_{4,0}, p_{4,1}\}$. Based on these definitions, EC-FRM-Code can be constructed by the following two steps.

Step-1: Identify All Groups. Each group and its corresponding elements can be identified by the following equations.

$$D_i = \{d_{\frac{i \cdot k}{n}, <i \cdot k>_n}, d_{\frac{i \cdot k+1}{n}, <i \cdot k+1>_n}, \cdots, d_{\frac{i \cdot k+k-1}{n}, <i \cdot k+k-1>_n}\} \quad (1)$$

Equation (1) illustrates the data elements of each group, which means all data elements are sequentially partitioned into $\frac{n}{r}$ groups. For example, when $i = 0, 1$, $D_0 = \{d_{0,0}, d_{0,1}, \cdots, d_{0,5}\}$ and $D_1 = \{d_{0,6}, d_{0,7}, \cdots, d_{1,0}, d_{1,1}\}$. We can easily find that the data elements of D_0 and D_1 are sequential.

$$P_{i,j} = \{(p_{\frac{k}{r}+j, <i \cdot k+k+j \cdot i>_n}, p_{\frac{k}{r}+j, <i \cdot k+k+j \cdot i+1>_n}, \cdots, p_{\frac{k}{r}+j, <i \cdot k+k+j \cdot i+r-1>_n}) \quad (0 \leqslant j \leqslant \frac{n-k}{r} - 1) \quad (2)$$

Equation (2) give the parity elements of $P_{i,j}$, which contains r elements due to the fact that $(i \cdot k+k+j \cdot i+r-1) - (i \cdot k + k + j \cdot i) + 1 = r$.

$$P_i = \bigcup_{j=0}^{\frac{n-k}{r}-1} P_{i,j} \quad (3)$$

$$G_i = D_i \cup P_i \quad (4)$$

Equation (3) and (4) give the definition of P_i and G_i, where P_i is composed of all $P_{i,j}$ and G_i is merged by D_i and P_i. Figure 4 shows an example of any (6,4) EC-FRM-Code (we use different icons to distinguish different groups). As it shows, each group contains n elements which are deployed in exact n different columns, because the column number of its related D_i and P_i are $\{< i \cdot k >_n, < i \cdot k + 1 >_n, \cdots, < i \cdot k + k - 1 >_n\}$ and $\{< i \cdot k + k >_n, < i \cdot k + k + 1 >_n, \cdots, < i \cdot k + n - 1 >_n\}$, respectively. For example, as shown in Figure 4, we can easily find that $D_0 = \{d_{0,0}, d_{0,1}, \cdots, d_{0,5}\}$, $P_{0,1} = \{p_{3,6}, p_{3,7}\}$ and $P_{0,2} = \{p_{4,8}, p_{4,9}\}$, where the column number of these elements are $\{0, 1, 2, \cdots, 9\}$. Obviously, they are deployed in different disks.

On the other hand, we can simply identify all the elements as follows.

1) Sequentially partition all data elements into D_i, where each D_i contains k sequential data elements. E.g., in the (6,2,2) EC-FRM-Code case shown in Figure 4, all data elements can be easily sequentially partitioned into D_0 to D_5, where each D_i is constituted by 6 data elements.
2) For each D_i, if the last element is $d_{x,y}$, we identify its related $P_{i,j}$ by $P_{i,0} = \{p_{\frac{k}{r}, <y+1>_n}, p_{\frac{k}{r}, <y+2>_n}, \cdots, p_{\frac{k}{r}, <y+r>_n}\}$, $P_{i,1} = \{p_{\frac{k}{r}+1, <y+r+1>_n}, p_{\frac{k}{r}, <y+r+2>_n}, \cdots, p_{\frac{k}{r}, <y+r+r>_n}\}$, $\cdots$, $P_{i, \frac{n-k}{r}-1} = \{p_{\frac{n}{r}-1, <y+(n-k-r)+1>_n}, p_{\frac{k}{r}, <y+(n-k-r)+2>_n}, \cdots, p_{\frac{k}{r}, <y+(n-k-r)+r>_n}\}$. E.g., as shown in Figure 4, the last data element of D_3 is $d_{2,3}$, then $P_{3,0} = \{p_{3,4}, p_{3,5}\}$ and $P_{3,1} = \{p_{4,6}, p_{4,7}\}$, because $r = 2$ and $\frac{k}{r} = 3$.
3) Merge all the elements of each D_i and $P_{i,j}$ $(0 \leqslant j \leqslant \frac{n-k}{r} - 1)$ into G_i.

Step-2: Construct over Each Group. After Step-1, each group (G_i) contains exact n elements distributed in n different columns, while all groups are independent of each other in encoding relationship. Therefore, we can logically consider each group as one stripe (row) of its candidate code, and then calculate all parity elements by the construction rules of the candidate code.

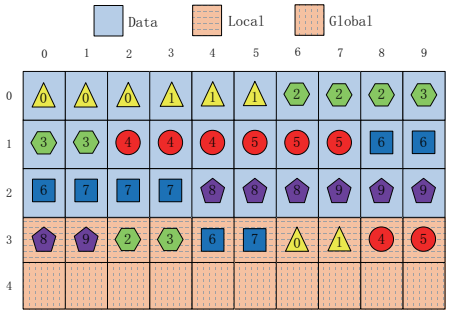
C. Fault Tolerance

EC-FRM-Code provides the same fault tolerance as its candidate code. I.e., for a specific EC-FRM-Code, if its candidate code can tolerate any f concurrent disk failures, the fault tolerance of this EC-FRM is exact f as well. We first give a lemma to assist the proof.

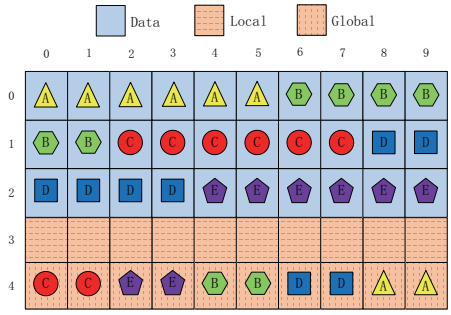
Lemma 1: For a given erasure code, switching any two elements of the same disk doesn't affect the fault tolerance.

Proof: Suppose that the two elements are $d_{i_1,j}$ and $d_{i_2,j}$ before switching, while assume they are $d'_{i_1,j}$ and $d'_{i_2,j}$ after being switched. Obviously, $d'_{i_1,j} := d_{i_2,j}$ and $d'_{i_2,j} := d_{i_1,j}$.

When some disks fail, the state of the jth disk may be either failed or survived. If failed, $d'_{i_1,j}$ and $d'_{i_2,j}$ are simultaneously lost before and after switching, thus we can reconstruct $d'_{i_1,j}$ and $d'_{i_2,j}$ following the method for recovering $d_{i_2,j}$ and $d_{i_1,j}$ before switching respectively; if survived, all lost elements are able to be recovered following the method before switching. Therefore, no matter the jth disk failed or survived, if we can recover it before switching, we can reconstruct all failed elements after switching as well. This feature assures that, switch any two elements of the same column doesn't affect the fault tolerance. □



(a) Local parity construction rules (Each parity element is calculated by the data elements with the same number of it. E.g., $p_{3,2}$ is calculated by $d_{0,6}$, $d_{0,7}$, and $d_{0,8}$).



(b) Global parity construction rules (Each parity element is computed by the data elements with the same letter of it. E.g., $p_{4,4}$ is computed by $d_{0,6}$, $d_{0,7}$, $d_{0,8}$, $d_{0,9}$, $d_{1,0}$, and $d_{1,1}$).

Figure 5: The construction rules of the (6,2,2) EC-FRM-LRC code.

According to Lemma 1, we can easily deploy all elements of each group in the same row by a series of switching two elements on the same disk, which doesn't affect the erasure code's fault tolerance. After switching, each row of EC-FRM-Code has the same layout and encoding rules as its candidate code, thus we can deduce that EC-FRM-Code have the same fault tolerance as its candidate codes as well.

D. Reconstruction

The reconstruction process of any EC-FRM-Code is straight-forward and can be briefly summarized as the following steps.

1) Start from the stripe level and identify the failed elements.
2) Move to the group level, and establish decoding equations based on the decoding relationships of its candidate code.
3) Solve the decoding equations to reconstruct the failed elements.

Now, we can reconstruct all failed elements from disk failures following above steps. In the next subsection, we will further discuss how to recover EC-FRM-Code from disk failures through a concrete case.

E. Case Study: LRC code in EC-FRM

We now discuss a case of EC-FRM-Code constructed by (6,2,2) LRC code. A stripe of (6,2,2) EC-FRM-LRC has 5 rows and 10 columns, where the first 3 rows lay data elements and the last 2 rows deploy parity elements. In order to facilitate our discussion, we simply lay all local parity elements in the $3th$ row, and deploy all global parity in the last ($4th$) row. Figure 5 shows the layout and construction rules.

1) Construction: As discussed above, groups in EC-FRM-Code are independent of each other in encoding relationship, thus we can easily calculate all parity elements based on the encoding equations of its candidate code. In this case, we denote the 10 elements of each group as $\{d_0, d_1, d_2, d_3, d_4, d_5, l_0, l_1, m_0, m_1\}$, and have the following encoding equations from LRC code [5],

$$l_0 = d_0 + d_1 + d_2 \tag{5}$$

$$l_1 = d_3 + d_4 + d_5 \tag{6}$$

$$m_0 = a_0 d_0 + a_1 d_1 + a_2 d_2 + b_0 d_3 + b_1 d_4 + b_2 d_5 \tag{7}$$

$$m_1 = a_0^2 d_0 + a_1^2 d_1 + a_2^2 d_2 + b_0^2 d_3 + b_1^2 d_4 + b_2^2 d_5 \tag{8}$$

For example, we can easily deduce that $G_1 = \{d_{0,6}, d_{0,7}, d_{0,8}, d_{0,9}, d_{1,0}, d_{1,1}, p_{3,2}, p_{3,3}, p_{4,4}, p_{4,5}\}$ (the green icons), where $\{d_{0,6}, d_{0,7}, d_{0,8}, d_{0,9}, d_{1,0}, d_{1,1}\}$ maps $\{d_0, d_1, d_2, d_3, d_4, d_5\}$, $\{p_{3,2}, p_{3,3}\}$ maps $\{l_0, l_1\}$, and $\{p_{4,4}, p_{4,5}\}$ maps $\{m_0, m_1\}$. Therefore, we can calculate $p_{3,2}$ by the equation of $p_{3,2} = d_{0,6} \oplus d_{0,7} \oplus d_{0,8}$ (as Figure 5(a) shows), and compute $p_{4,4}$ by the equation of $p_{4,4} = a_0 d_{0,6} + a_1 d_{0,7} + a_2 d_{0,8} + b_0 d_{0,9} + b_1 d_{1,0} + b_2 d_{1,1}$ (as shown in Figure 5(b)). Similarly, we can encode other groups following this method, until all parity elements of the stripe have been computed.

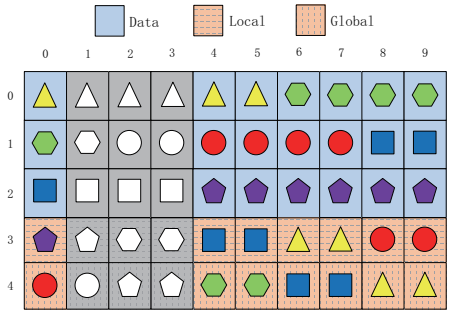
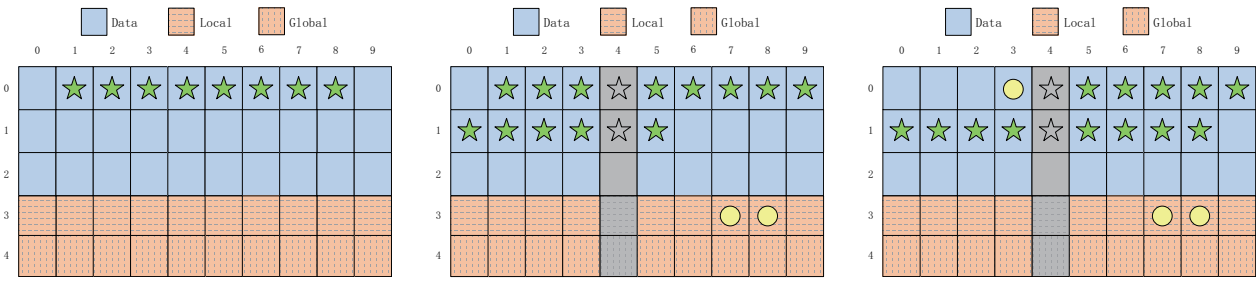


Figure 6: An example of reconstruction from disk $1, 2, 3$ concurrent failures in (6,2,2) EC-FRM-LRC.



(a) An 8-elements normal read. (b) A 14-elements degraded read. (c) Another 14-elements degraded read.

Figure 7: The read features of (6,2,2) EC-FRM-LRC code (The star icons indicate the data elements that need to be read, while the round icons represent the elements that need to be extra read).

2) Reconstruction: We now discuss how EC-FRM-LRC addresses disk failures by analyzing an example shown in Figure 6. In this case, we can easily establish the related decoding equations after identifying the failed elements in each group. For example, let's consider G_3 (identity by square icons), where the failed elements are $d_{2,1}$ (d_3), $d_{2,2}$ (d_4), and $d_{2,3}$ (d_5). We can directly establish the following decoding equations based on Equation (6)-(8).

$$d_3 + d_4 + d_5 = l_1 \tag{9}$$

$$b_0 d_3 + b_1 d_4 + b_2 d_5 = m_0 + a_0 d_0 + a_1 d_1 + a_2 d_2 \tag{10}$$

$$b_0^2 d_3 + b_1^2 d_4 + b_2^2 d_5 = m_1 + a_0^2 d_0 + a_1^2 d_1 + a_2^2 d_2 \tag{11}$$

Since the right-hand side of above three equations are constants, it is easy to deduce that the coefficients matrix of the unknown variables on the left-hand is as follows:

$$G = \begin{pmatrix} 1 & 1 & 1 \\ b_0 & b_1 & b_2 \\ b_0^2 & b_1^2 & b_2^2 \end{pmatrix} \tag{12}$$

According to [5], (6,2,2) LRC code can be recovered from any kinds of triple disk failures, thus $Det(G) \neq 0$ and we can recalculate the failed elements based on Equation (9)(10)(11). Similar, we can recover the failed elements in other groups following this method, until all lost elements are reconstructed.

V. Properties

In this section, we analyze some important properties for EC-FRM-Code, including read performance, fault tolerance, storage overhead, and the number of disks that can be applied to. The goal of our analysis is to illustrate that EC-FRM-Code not only performs well on both normal reads and degraded reads, but also relaxes the restrictions that existing vertical codes confront.

A. The Read Properties

We start with some read examples over (6,2,2) EC-FRM-LRC code, in order to explain how I/O accesses distribution under read operations.

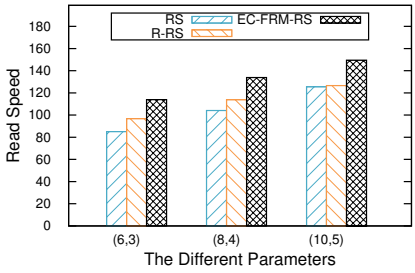
Normal Reads: Since EC-FRM-Code sequentially deploys data elements among all disks, the code will provide good performance on normal reads. For example, as shown in Figure 7(a), the most loaded disk in EC-FRM-LRC code just needs to read only one element for the 8-elements' read operation, while it needs to read two elements in LRC code with both standard form and rotated stripes (as shown in Figure 3(a) and 3(b)). Therefore, EC-FRM-Code will provide better performance than its candidate codes due to the fact that it provides more evenly distribution on normal reads.

Degraded Reads: The feature on degraded reads is more complex than that on normal reads. E.g., let's first observe the Figure 7(b), where the most loaded disk only needs to afford two elements' read accesses. For standard LRC code, the most loaded disk needs to read at least 3 elements, because the total number of read accesses is no less than 14 and there are just 6 disks (5 data disks and 1 local parity disk) contributed to degraded reads. However, things are not always fine. When it comes to the example shown in Figure 7(c), the system also needs to read 14 elements in total, but the most loaded disk needs to read 3 elements, thus could not improve the performance. In summary, EC-FRM-Code could improve the degraded read performance by more evenly distribution, but the improved range will be less than that on normal reads.

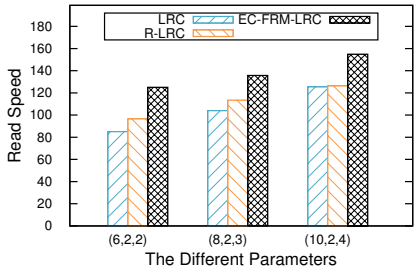
B. Other Outstanding Properties

EC-FRM-Code provides some outstanding properties concerned by cloud storage system designers, which eliminates weakness of vertical codes mentioned in [3].

Adapter for Arbitrary Number of Disks: From the construction steps in Section IV-B, EC-FRM-Code can be constructed upon any parameter of its candidate code, and the total number of disks in EC-FRM-Code is the same as



(a) The speed for RS code (unit: MB/s).



(b) The speed for LRC code (unit: MB/s).

Figure 8: The normal read results for tested codes with different forms.

its candidate code. Since the candidate codes are horizontal codes which can be applied to arbitrary number of disks, EC-FRM-Code can be applied to arbitrary number of disks as well.

High Fault Tolerance and Low Storage Overhead: As referred in Section IV-C, EC-FRM-Code and its candidate code provide the same fault tolerance. Similarly, since EC-FRM-Code is constructed by redeploying the data and parity distribution over its candidate code, the storage overhead of EC-FRM-Code and its candidate code should be the same as well. Therefore, since candidate codes like Reed-Solomon code and LRC code can be designed to tolerate arbitrary number of disk failures with low storage overhead, EC-FRM-Code achieves both low storage overhead and high fault tolerance as well.

VI. Experiment Evaluations

We select Reed-Solomon code and LRC code as candidate codes to evaluate our proposed EC-FRM, because these codes are widely used in real storage systems [5][12]. We mirror the codes and parameters in Table I. For each code, we implement the code with standard form, the code with rotated stripes, and the code with EC-FRM form (i.e., its related EC-FRM-Code) in a real system based on Jerasure-1.2 library [21], which is an open source library and commonly used in erasure code community [22].

Table I: The tested erasure codes and paraments

RS Code	LRC Code
(6,3)	(6,2,2)
(8,4)	(8,2,3)
(10,5)	(10,2,4)

A. Experiment Environment

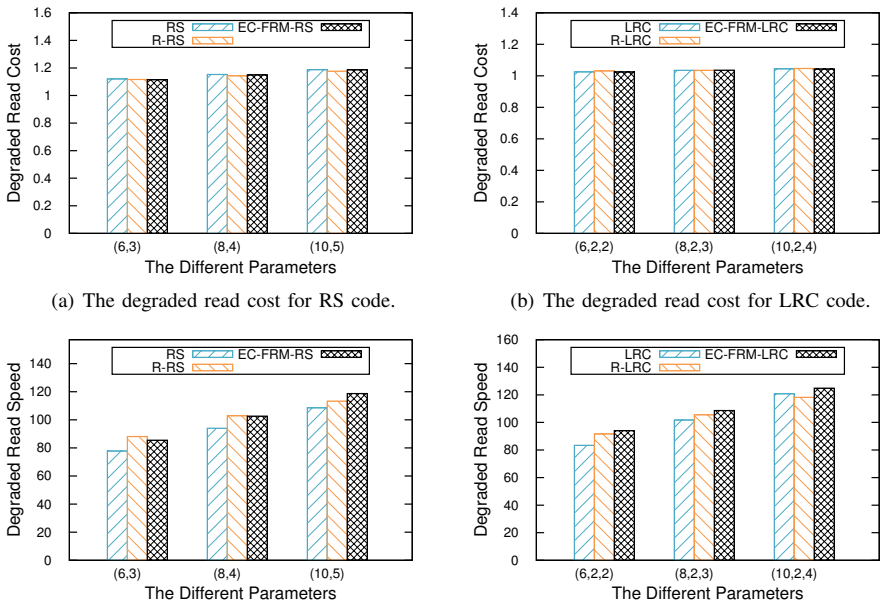
Due to the restriction on our experiment platform, we run all experiments on a machine and a disk array. The similar experiment environment has been widely used as testbed for cloud storage systems [3]. The tested machine has an Intel Xeon X5472 processor and 12 GB of RAM. The disk array is constituted with 16 disks, where the type of each disk is Seagate/Savvio 10K.3 and the model number is ST9300603SS. Each disk has 300GB capability and 10000 rpm. The operation system of the machine is SUSE with Linux fs91 3.2.16.

B. The Experiments on Normal Reads

We run 2000 times of experiment for Reed-Solomon code and LRC code with standard form, rotating form, and EC-FRM form, respectively. Each time of experiment we randomly generate the start point and the read size, where the start point may be an arbitrary data element and the range of read size is 1 to 20 data elements. We record the real read speed in each time of experiment, and calculate the average value as our evaluation metric, in order to compare the real performance among the tested codes with different configurations.

Figure 8 shows the read speed for various codes with different parameters. Let's first consider Reed-Solomon code. As Figure 8(a) shows, compared to standard Reed-Solomon code, EC-FRM-RS achieves 19.2% to 33.9% higher read speed, because all disks in EC-FRM-RS contribute to read operations while only a part of disks (i.e., k disks) in standard RS code help to reads. The more number of contributed disks will decrease the possibility of the needed elements placed in the same disk, thus minimize the loads on the most loaded disk and improve the speed. Similarly, as shown in Figure 8(b), EC-FRM-LRC gains 23.5% to 46.9% higher read speed than standard LRC code.

On the other hand, though rotating the mappings from logic disks to physical disks stripe by stripe could improve the read speed in some level, the codes with rotated stripes still provide much lower speed than EC-FRM-Code. In statistics, EC-FRM-RS code achieves 17.7% to 18.1% higher read speed than Reed-Solomon code with rotated stripes, while EC-FRM-LRC achieves 19.6% to 29.3% higher speed than LRC code with rotated stripes. All these results illustrate that EC-FRM-Code achieves good performance on normal reads.



(a) The degraded read cost for RS code.

(b) The degraded read cost for LRC code.

(c) The speed for RS code (unit: MB/s).

(d) The speed for LRC code (unit: MB/s).

Figure 9: The degraded read results for tested codes with different forms.

C. The Experiments on Degraded Reads

We then conduct 5000 times of experiment to evaluate the performance on degraded reads. For each time of experiment, we randomly generate the start point, the erased disk, and the read size, where the start point may be an arbitrary data element, the erasure disk maybe an arbitrary disk, and the range of read size is 1 to 20 data elements. We select both degraded read cost and degraded read speed as evaluation metrics, because the cost represents the usage of network bandwidth while the speed directly reflects the performance. In addition, we calculate the average cost the the average speed among the evaluations for comparison.

The Degraded Read Cost: Figure 9(a) and 9(b) show the degraded read cost among the tested configurations. As they show, the distinctions between the different forms of both Reed-Solomon code and LRC code are very tiny: for Reed-Solomon code, the difference between various forms is less than 0.9%, while for LRC code the difference is even less than 0.7%. On the other hand, the degraded read cost for LRC code is much less than that in Reed-Solomon code, because LRC code trades the storage overhead for reducing the degraded read cost. In summary, EC-FRM-RS code and EC-FRM-LRC code provide very close degraded read cost with other forms of Reed-Solomon code and LRC code, respectively.

The Degraded Read Speed: We show the degraded read speed results in Figure 9(c) and 9(d). As the figure shows, EC-FRM-RS code achieves 9.1% to 9.9% higher speed than standard Reed-Solomon code, while EC-FRM-LRC code gains 3.3% to 12.8% higher degraded read speed than standard LRC code.

However, Reed-Solomon code and LRC code with rotated stripes also provide good performance on degraded reads, because the rotated stripes increase the possibility of the needed elements placed in different disks. Compared to Reed-Solomon code with rotated stripes, EC-FRM-RS achieves 4.7% higher degraded read speed when $k = 10$, but provides 0.26% and 2.9% lower degraded read speed when $k = 8$ and $k = 6$, respectively. Since the difference is small and degraded reads only appear when some disks become unavailable, the impact of such a slight degraded speed can be ignored in cloud storage systems. On the other hand, EC-FRM-LRC provides 2.6%, 2.9%, and 5.7% higher speed than LRC code with rotated stripes when $k = 6, 8, 10$, respectively.

In summary, EC-FRM-RS gains higher degraded read speed than standard Reed-Solomon code, achieves a little higher speed than Reed-Solomon code with rotated stripes in some cases, but provides a little slower speed than Reed-Solomon code with rotated stripes under the other cases; EC-FRM-LRC code gains much higher degraded read speed than both standard LRC code and LRC code with rotated stripes. All in all, combined with high read speed on normal reads, we can conclude that EC-FRM-Code provides high performance on read operations.

VII. Conclusion

In this paper, we propose a novel erasure coding framework named EC-FRM, in order to integrate existing codes

to improve the read performance for cloud storage systems. The integrated erasure code named EC-FRM-Code, which is constructed by redeploying the layout of the data elements in its candidate code (maybe Reed-Solomon code or LRC code). EC-FRM-Code not only achieves good performance on both normal reads and degraded reads, but also keeps most of wonderful properties of its candidate code. E.g., EC-FRM-RS code provides the MDS properties and can tolerate arbitrary number of disk failures, while EC-FRM-LRC code significantly reduce the I/O accesses on degraded reads. The experiment results show that EC-FRM-RS gains 19.2% to 33.9% higher normal read speed and 9.1% to 9.9% higher degraded read speed than standard Reed-Solomon code, while EC-FRM-LRC code owns 23.5% to 46.9% higher normal read speed and 3.3% to 12.8% higher degraded read speed than standard LRC code.

ACKNOWLEDGMENT

We would like to greatly appreciate the anonymous reviewers for their insightful comments. This work is supported by the National Natural Science Foundation of China (Grant No. 61232003, 61327902), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), the State Key Laboratory of High-end Server and Storage Technology (Grant No.2014HSSA02).

REFERENCES

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

[2] D. Borthakur, et al. HDFS RAID. Hadoop User Group Meeting, November, 2010.

[3] O. Khan, R. Burns, J. Plank, and W. Pierce. Rethinking erasure codes for cloud file systems: minimizing I/O for recovery and degraded reads. In Proceedings of USENIX FAST'12, February 2012.

[4] J. Plank, E. Miller, K. Greenan, B. Arnold, J. Burnum, A. Disney, and A. McBride. GF-Complete: A comprehensive open source library for galois field arithmetic. Version 1.0. Technical Report UT-CS-13-716, September, 2013.

[5] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In Proceedings of USENIX ATC'12, June, 2012.

[6] I. Reed and G. Solomon. Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematice, pages 300-304, 1960.

[7] J. Hafner. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In Proceedings of USENIX FAST'05, December, 2005.

[8] L. Xu, and J. Bruck. X-Code: MDS array codes with optimal encoding. IEEE Transaction on Information Theory, 45(1):272-276, January 1999.

[9] B. Schroeder and G. Gibison. Disk failures in the read world: What does an MTTF of 1,000,000 mean to you? In Proceedings of USENIX FAST'07, 2007.

[10] J. L. Hafner, V. Deenadhayalan, T. Kanungo, and K. Rao. Performance metrics for erasure codes in storage systems. Technical Report, RJ 10321, IBM Research, San Jose, 2004.

[11] E. Pinheiro, W. Weber, and L. Barroso. Failure trends in a large disk drive population. In Proceedings of the USENIX FAST'07, February 2007.

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

[13] F. MacWilliams and N. Sloane. The Theroy of Error Correcting Codes. Amsterdam: North-Holland, 1977.

[14] P. Corbett, B. English, A. Goel, T. Grcanac, S.Kleiman, J. Leong, and S. Sankar. Row-Diagonal Parity for double disk failure correction. In Proceedings of the USENIX FAST'04, San Francisco, March 2004.

[15] J. Plank. The RAID-6 liberation codes. In Proceedings of the USENIX FAST'08, February 2008.

[16] J. Blomer, M. Kalfane, R. Krap, M. Karpinski, M. Luby, and D. Zuckerman. An XOR-based Erasure-Resilient coding scheme. Technical Report TR-95-048, International Computer Science Institute, 1995.

[17] C. Wu, X. He, G. Wu, S. Wan, X. Liu, Q. Cao, and C. Xie. HDP Code: A horizontal diagonal parity code to optimize I/O load balance in RAID-6. In Proceedings of IEEE/IFTP DSN'11, Hong Kong, June 2010.

[18] J. Plank, M. Blaum, and J. Hafner. SD Codes: Erasure Codes Designed for How Storage Systems Really Fail. In Proceedings of the USENIX FAST'13, February 2013.

[19] M. Li and P. Lee. STAIR Codes: A General Family of Erasure Codes for Tolerating Device and Sector Failures in Practical Storage Systems. In Proceedings of the USENIX FAST'14, 2014.

[20] C. Huang and L. Xu. STAR: an efficient coding scheme for correcting triple storage node failures. In Proceedings of USENIX FAST'05, 2005.

[21] J. Plank, S. Simmerman, and C. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications-Version 1.2. Technical Report CS-08-627, University of Tennessee, August, 2008.

[22] J. Plank, J. Luo, C. Schuman, L. Xu and Z. W. O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In Proceedings of USENIX FAST'09, February, 2009.

[23] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. Oceanstore: An architecture for global-scale persistent storage. In Proceedings of ASPLOS'00, 2000.

[24] M. Li and J. Shu. On Cyclic Lowest Density MDS Array Codes Constructed Using Starters. In Proceedings of IEEE ISIT'10, June, 2010.

[25] Y. Fu and J. Shu. D-Code: An Efficient RAID-6 Code to Optimize I/O Loads and Read Performance. In Proceedings of IEEE IPDPS'15, May, 2015.

[26] Z. Shen and J. Shu. HV-Code: An All-around MDS Code to Improve Efficiency and Reliability of RAID-6 Systems. In Proceedings of DSN'14, 2014.

[27] L. Xiang, Y. Xu, J. Lui, and Q. Chang. Optimal recovery of single disk failures in RDP code storage systems. In Proceedings of the ACM SIGMETRICS'10, New York, 2010.

[28] Y. Fu, J. Shu, and X. Luo. A Stack-Based Single Disk Failure Recovery Scheme for Erasure Coded Storage Systems. In Proceedings of IEEE SRDS'14, October, 2014.

[29] J. Li, S. Yang, X. Wang, and B. Li. Tree-Structed Data Regeneration in Distributed Storage Systems with Regenerating Codes. In Proceedings of IEEE INFOCOM'10, 2010.

[30] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In Proceedings of USENIX OSDI'08, 2008.