2015 IEEE 29th International Parallel and Distributed Processing Symposium

D-Code: An Efficient RAID-6 Code to Optimize I/O Loads and Read Performance

Yingxun Fu and Jiwu Shu*
Tsinghua National Laboratory for Information Science and Technology
Department of Computer Science and Technology, Tsinghua University
Beijing, China, 100084
**Corresponding author: shujw@tsinghua.edu.cn*
fu-yx10@mails.tsinghua.edu.cn

***Abstract*—With the reliability requirement increasingly important, RAID-6, which can tolerate any two concurrent disk failures, has been widely used in various storage systems. One class of typical RAID-6 implementations is to use Maximum Distance Separable (MDS) erasure codes. However, most existing RAID-6 MDS codes suffer from unbalanced I/O or high I/O cost, and cannot provide satisfied read performance on both normal mode and degraded mode. All these metrics are important in modern storage systems.**

In this paper, we propose a new RAID-6 MDS code termed D-Code to address these problems. D-Code uses a new kind of horizontal parities to increase the possibility of continuous data elements sharing the common parities in order to provide low I/O cost and good degraded read performance, while uses deployment parities to assure all parities can be evenly distributed in the last two rows in order to achieve good load balancing and good normal read performance. Our evaluations and experiments show that D-Code not only provides good load balancing and low I/O cost under different workloads, but also achieves good performance on both normal reads and degraded reads. E.g., D-Code achieves up to 21.3% and 13.5% higher read speed than RDP code and H-Code in normal mode, while gains up to 26.0% higher read speed than X-Code in degraded mode.

***Keywords*-RAID-6; Erasure Code; Load Balancing; I/O Cost; Normal Read; Degraded Read;**

I. Introduction

Modern storage systems, such as GFS [1] and Windows Azure [2], usually host their data in a large number of storage devices. However, the mass of data disks' equipment will in turn increase the probability of data loss or damage, because of the appearance of various kinds of disk failures. To ensure the reliability of the hosted data, storage systems need to tolerate the disk failures and recover the lost information. In recent years, RAID-6, which distributes data across different disks and supports parallel operations like reads and writes, has received a lot of attentions due to the fact that it can tolerate any two concurrent disk failures.

There are a number of candidates for implementing RAID-6, where Maximum Distance Separable (MDS) [3] code which aims to protect data against disk failures with the optimal storage rate is a classic kind of RAID-6 implementation and gains a lot of attentions. Based on the parity distribution, RAID-6 MDS codes can be simply classified as horizontal codes and vertical codes. Horizontal codes, such as RDP code [4] and EVENODD code [5], are composed of $k+2$ disks, where the first k disks host user's data and the last 2 disks keep the parities. Vertical codes, such as X-Code [6] and H-Code [7], distribute their parities among all disks.

However, most existing RAID-6 MDS codes cannot provide satisfied I/O loads under two crucial metrics: the load balancing and the I/O cost (i.e., the total amount of elements that need to be read/written). For horizontal codes, the two parity disks don't contribute to reads and suffer from a lot of accesses over partial stripe writes, which causes unbalanced I/O. Some vertical codes like H-Code [7] also suffer from this problem, because they also have one specialized parity disk. Though other vertical codes such as X-Code [6] evenly distributing all parities among disks provide good load balancing, they usually suffer from high I/O cost, because they don't adopt row parity and partial stripe writes usually need to write some continuous data elements located on the same row. In addition, some global load balancing methods such as rotating the mappings from logic disks to physical disks stripe by stripe [8][9] may alleviate the unbalanced I/O in some level, but they cannot balance the I/O accesses on the same stripe, thus cannot solve this problem due to the fact that each stripe has different access frequencies [17].

On the other hand, the performance on both normal reads and degraded reads are other essential metrics, because these operations are very frequently occurred in today's storage systems. However, most existing RAID-6 codes cannot provide satisfied read performance. For horizontal codes, the parity disks don't help to normal reads degrading the read performance, because all disks in RAID system can be accessed in parallel [10]. Though some vertical codes like X-Code achieve good performance on normal reads, they cannot provide satisfied degraded read performance, because the continuous data elements that need to be read usually share the common horizontal parities (i.e., row parities) rather the same diagonal parities.

In this paper, we propose a novel RAID-6 MDS code named Deployment Code (D-Code), in order to achieve both good load balancing and low I/O cost and provide good performance on normal reads and degraded reads, which are important in read-only workloads (cloud storage systems), read-intensive workloads (dependable SSD arrays), and read-write evenly mixed workloads (traditional file systems over dependable disk arrays). A stripe of D-Code can be represented as an n-rows and n-columns matrix, where n needs to be a prime number. Different from previous RAID-6 MDS codes, D-Code uses horizontal parities and deployment parities (which are a special kind of diagonal parities that the related data elements are deployed one by one based on a special deployment method), in order to assure continuous data elements more likely sharing the common horizontal parities and to keep all parities evenly distributed in the last two rows. The increased probability of continuous data elements sharing the common horizontal parities reduces the I/O cost on partial stripe writes and improves the degraded read performance, while the parity distribution makes D-Code achieve good load balancing and good normal read performance. All in all, our contributions can be summarized as follows:

- We propose a novel RAID-6 MDS erasure code termed D-Code, in order to optimize the I/O loads and to improve the read performance.
- We conduct a series of analysis on the representative features of our D-Code. The results indicate that D-Code achieves the optimal storage efficiency, encoding/decoding computational complexity, update complexity, and good performance on single disk failure recoveries.
- We build a number of simulations to evaluate the I/O loads in D-Code. The results illustrate D-Code not only gains good load balancing, but also achieves lower I/O cost compared to other well-balanced codes under different workloads.
- We implement D-Code based on Jerasure 1.2 [11], and evaluate its read performance in a real storage system. The experiment results show that, when compared to other popular RAID-6 MDS codes, D-Code achieves better read speed on both normal mode and degraded mode. For example, D-Code achieves up to 21.3% and 13.5% higher read speed than RDP code and H-Code in normal mode, while gains up to 26.0% higher read speed than X-Code in degraded mode.

II. Problem Statement and Motivation

A. The I/O Load Problem in Existing RAID-6 Codes

Load Balancing: Load balancing is a classic performance metric and has attracted a lot of attentions in recent years, because an unbalanced I/O load may extend the operation time and even hurt the reliability by causing uneven I/O cost to disks [12][13][14]. However, most RAID-6 codes cannot provide good load balancing. E.g., horizontal codes like RDP code [4] have two specialized parity disks don't contribute to reads and encounter more I/O accesses than data disks over partial stripe writes (as Figure 1(b) shows), thus suffer from unbalanced I/O. Though many dynamic load balancing approaches [15][16] and static rotating the mappings from logic disks to physical disks stripe by stripe like RAID-5 [8] may alleviate this problem in some level, they either have additional overheads for monitoring the status of disks or also suffer from unbalanced I/O due to the fact that the access frequencies of each stripe are different. Some vertical codes like H-Code and P-Code also suffer from this problem, because they either have specialized parity disks or unevenly distribute the parities. Though another kind of vertical codes like X-Code [6] and HDP code [17] achieve good load balancing, they usually encounter much higher I/O cost on partial stripe writes.

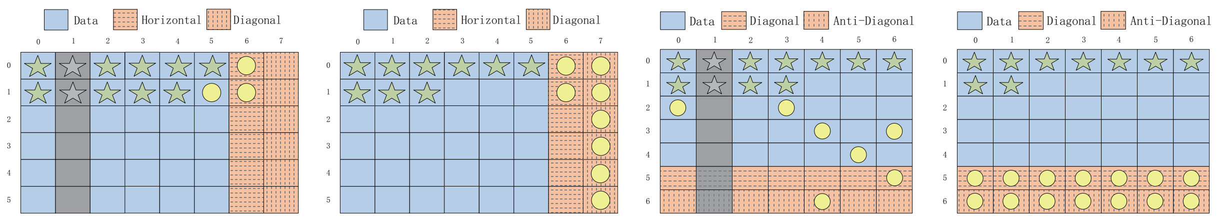
I/O Cost: The I/O cost, which indicates the total I/O accesses among all disks, is another important performance metric in storage systems, because high I/O cost may increase disks' I/O burden and degrade the performance. However, most well-balanced codes suffer from high I/O cost on partial stripe writes (i.e., write to a series of continuous data elements). For example, X-Code (which uses diagonal parities and anti-diagonal parities evenly distributed among all disks) is a well-balanced code, but it suffers from higher I/O cost than horizontal codes under partial stripe writes, because the continuous data elements (that need to be written) usually share the common horizontal parities rather than diagonal parities (as shown in Figure 1(d) and 1(b)).

In summary, horizontal codes and a part of vertical codes cannot provide good load balancing, while other vertical codes suffer from high I/O cost. If there exists a RAID-6 code achieves both good load balancing and low I/O cost, it will provide better I/O loads than existing RAID-6 codes.

B. The Read Problem in Existing RAID-6 Codes

Reads are crucial operations in modern storage systems, because they are occurred very frequently. There are two important kinds of read operations: the normal read operations (read to continuous data elements without disk failures) and degraded read operations (read to both available data elements and temporary unavailable data elements under disk failures). However, most RAID-6 codes cannot provide satisfied performance on read operations.

For horizontal codes such as RDP code [4], the parity disks don't contribute to normal reads degrading the performance, because disks in RAID system can be accessed in parallel. Though some vertical codes like X-Code provide good normal read performance due to the fact that all disks in these codes can help to normal reads, they cannot provide good degraded read performance, because the continuous data elements (that need to be read) usually share the same horizontal parities rather than diagonal parities (as Figure 1(a) and Figure 1(c) shows). Moreover, though other vertical



(a) Degraded read in RDP code. (b) Partial stripe write in RDP code. (c) Degraded read in X-Code. (d) Partial stripe write in X-Code.

Figure 1: Examples of degraded reads and partial stripe writes in RDP code and X-Code when $p = 7$ (The star icons indicate the data elements that need to be read/written, while the round icons represent the elements that need to be extra read/written).

codes like HDP code [17] and H-Code [7] have horizontal parities, they either have specialized parity disks or distribute parities in the middle of stripe, which degrade the normal read performance. Therefore, if we can design a code to provide good read performance on both normal mode and degraded mode, the designed code will provide better read performance than existing RAID-6 codes.

C. Our Motivation

In order to overcome the shortcomings in existing codes discussed above, we should design a new code with properties of good load balancing, low I/O cost, and good performance on both normal reads and degraded reads. The motivation of this paper is as follows.

Evenly Distribute Parities among All Disks: As discussed above, unbalanced I/O is mainly caused by parities' uneven distribution. This observation motivates us to evenly distribute the parities among all disks.

Separate Data and Parities in Different Rows: In order to enlarge the number of disks that can contribute to normal reads, we need to lay the data elements in some continuous rows, while deploy the parity elements in the other rows. Therefore, the second motivation of us is to separate data elements and parity elements in different rows.

Increase The Possibility of Continuous Data Elements Sharing The Common Parities: X-Code satisfies all motivations mentioned above and provides good load balancing and normal read performance, but it suffers from high I/O cost on partial stripe writes and degraded reads, because continuous data elements are hard to share the same diagonal parities. This observation motivates us to design new encoding rules for increasing the possibility of continuous data elements sharing the common parities.

III. D-Code

Based on the motivations referred above, in this paper we propose a new RAID-6 MDS code named Deployment Code (D-Code). Different from previous RAID-6 MDS codes, D-Code uses horizontal parities and deployment parities, where each horizontal parity is calculated by the XOR sum of some continuous data elements. The deployment parities are a special kind of diagonal parities that the related data elements are deployed one by one based on a special deployment method. We now discuss the detailed designs of D-Code.

A. Data/Parity Layout and Encoding Rules

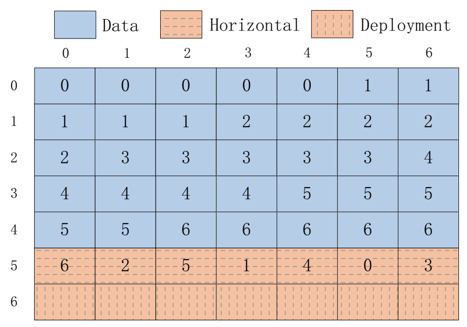
A stripe of D-Code can be represented as an n-rows and n-columns matrix, which contains three kinds of elements: data elements, horizontal parity elements and deployment parity elements, n needs to be a prime number. In the matrix, data elements are laid in first $n-2$ rows, while parity elements are deployed in the last 2 rows. We use $\boldsymbol{D_{i,j}}$ ($0 \leqslant i \leqslant n-3, 0 \leqslant j \leqslant n-1$) to denote the ith data element of the jth column, while use $\boldsymbol{P_{i,j}}$ ($n-2 \leqslant i \leqslant n-1$, $0 \leqslant j \leqslant n-1$) to denote the ith parity element of the jth column.

Horizontal Encoding Rules: The horizontal encoding rules of D-Code can be represented in mathematics as the following equation (mathematical notation $\bigoplus$ and $<>$ indicate XOR operation and modular arithmetic operation, respectively).

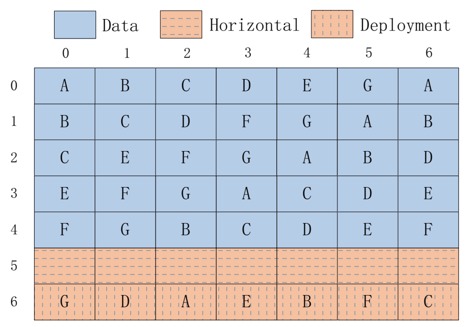
$$P_{n-2,i} = \bigoplus_{j=0}^{n-3} D_{<\frac{n-3}{2}\cdot(<i+j+2>_n - j)>_{n-2}, <i+j+2>_n}, \quad (0 \leqslant i \leqslant n-1) \tag{1}$$

In order to clarify the physical meaning of horizontal encoding rules, we define **the next horizontal element of $\boldsymbol{D_{i,j}}$** as: if $j \neq n-1$, the next element is $D_{i,j+1}$; otherwise, the next element is $D_{i+1,0}$. In simple terms, if an element ($D_{i,n-1}$) reaches the end of a row, the next horizontal element will be the first element of the next row ($D_{i+1,0}$). Based on this definition, the horizontal encoding rules can be represented as 4 steps:

1) **Assign an identification number for each data element:** We denote $D_{0,0}$ as the $0th$ horizontal element, the next horizontal element of $D_{0,0}$ (i.e., $D_{0,1}$) as the $1th$ element, then the next ($D_{0,2}$) as $2th$ element, until the last data element ($D_{n-3,n-1}$) has been denoted.



(a) Horizontal encoding rules



(b) Deployment encoding rules.

Figure 2: An example of D-Code with 7-disks.

2) **Label each data element by a number flag:** We label the $0th$ to $(n-3)th$ element as number '0', the $(n-2)th$ to $((n-2)+(n-3))th$ element as number '1', and so on.
3) **Label each horizontal parity element by a number flag:** For each $0 \leqslant k \leqslant n-1$, we suppose the $(k \cdot (n-2)+(n-3))th$ element is $D_{x,y}$ and label the parity element $P_{n-2,<y+1>_n}$ as the same number of the $(k \cdot (n-2)+(n-3))th$ element's.
4) **Compute all horizontal parities:** For each labeled number, we calculate the XOR sum of its corresponding data elements and store the sum in its related parity element.

Figure 2(a) shows an example of the horizontal encoding rules in 7-disks D-Code. We can easily observe that the $10th$ to $14th$ (i.e., $(2 \cdot (n-2))th$ to $(2 \cdot (n-2)+(n-3))th$) horizontal elements are $D_{1,3}$, $D_{1,4}$, $D_{1,5}$, $D_{1,6}$, $D_{2,0}$, where their corresponding parity element is $P_{5,1}$ in terms of $<0+1>_7 = 1$. Therefore, we label these elements as the same number '2', calculate the XOR sum of the data elements and store the sum in the parity element, i.e., $P_{5,1} = D_{1,3} \oplus D_{1,4} \oplus D_{1,5} \oplus D_{1,6} \oplus D_{2,0}$.

Deployment Encoding Rules: The deployment encoding rules can be represented in mathematics as the following equation.

$$P_{n-1,i} = \bigoplus_{j=0}^{n-3} D_{<\frac{n-3}{2}\cdot(<i-j-2>_n - j)>_{n-2},<i-j-2>_n}, \quad (0 \leqslant i \leqslant n-1) \tag{2}$$

In order to illustrate the implication of deployment rules, we define **the next deployment element of $D_{i,j}$** as: if $j \neq 0$, the next element is $D_{<i+1>_{n-2},j-1}$; otherwise, the next element is $D_{i,n-1}$. In other word, if an element is not located on the first column, the next deployment element is the element on the below left of it; otherwise, the next deployment element is the last element of the current row (the ith row). Similar as horizontal encoding rules, our deployment encoding rules can be represented as follows.

1) **Assign an identification number for each data element:** We denote $D_{0,0}$ as the $0th$ deployment element, the next deployment element of $D_{0,0}$ (i.e., $D_{0,n-1}$) as the $1th$ element, then the next (i.e., $D_{1,n-2}$) as $2th$ element, until the last data element ($D_{n-3,1}$) has been denoted.
2) **Label each data element by a letter flag:** We label the $0th$ to $(n-3)th$ element as letter 'A', the $(n-2)th$ to $((n-2)+(n-3))th$ element as letter 'B', and so on.
3) **Label each deployment parity element by a letter flag:** We label the $2th$ deployment parity element ($D_{6,2}$) as letter 'A', the $<4>_n$ th deployment parity element as letter 'B', $\cdots$, until the $<2n>_n$ th (i.e., $0th$) deployment parity element has been labeled.
4) **Compute all deployment parities:** For each labeled letter, we calculate the XOR sum of its corresponding data elements and store the sum in its related parity element.

Figure 2(b) illustrates an example of the deployment encoding rules in 7-disks D-Code. It is easy to see that the $0th$ to $4th$ (i.e., $(n-3)th$) deployment elements are $D_{0,0}$, $D_{0,6}$, $D_{1,5}$, $D_{2,4}$, and $D_{3,3}$, where their corresponding parity element is $P_{6,2}$ in terms of $<2>_7 = 2$. Therefore, we label these elements as letter 'A', calculate the XOR sum of the data elements and store the sum in the parity element, i.e., $P_{6,2} = D_{0,0} \oplus D_{0,6} \oplus D_{1,5} \oplus D_{2,4} \oplus D_{3,3}$.

B. Correctness Proof

In this subsection, we prove the correctness of D-Code in one stripe, because all stripes are dependent on each other in encoding/decoding relationships. We first give two lemmas to illustrate that reordering each column's elements doesn't affect the erasure code's fault-tolerance, and then prove that D-Code can be constructed by reordering the elements of each column of X-Code.

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

Proof: We first suppose the two switched elements are $E_{i_1,j}$ and $E_{i_2,j}$ before switching, while assume they are $N_{i_1,j}$ and $N_{i_2,j}$ after switching. Obviously, $E_{i_1,j} \to N_{i_2,j}$ and $E_{i_2,j} \to N_{i_1,j}$.

When some disks fail, the status of jth disk maybe either failed or survived. If failed, $N_{i_1,j}$ and $N_{i_2,j}$ are simultaneously broken, thus we can reconstruct $N_{i_1,j}$ by the method for recovering $E_{i_2,j}$ before switching, while reconstruct $N_{i_2,j}$ by the method for recovering $E_{i_1,j}$ before switching, because $E_{i_1,j}$ and $E_{i_2,j}$ are simultaneously broken before switching as well; If survived, we can reconstruct all lost elements by the recovery method before switching. Therefore, for any failure situations, if we can recover it before switching, then we can reconstruct it after switching as well. This feature assures that switch any two elements of the same column doesn't affect erasure code's fault-tolerance. □

Lemma 2: For a given erasure code, reordering the elements of each column doesn't affect the fault-tolerance.

Proof: It is easy to deduce that reordering operation can be split into a series of switching operations, thus doesn't affect the fault-tolerance (according to Lemma 1). □

Theorem 1: D-Code can be generated by reordering the elements of each column of X-Code under the following equation,

$$E_{i,j} := \begin{cases} N_{<\frac{n-3}{2}(j-i)>_{n-2},j}, & (0 \leqslant i \leqslant n-3) \\ N_{i,j}, & (n-2 \leqslant i \leqslant n-1) \end{cases} \tag{3}$$

where $E_{i,j}$ denotes the ith element of the jth column in X-Code before reordering, $N_{i,j}$ means the ith element of the jth column after reordering, $0 \leqslant j \leqslant n-1$.

Proof: The encoding rules of X-Code before reordering can be represented as the following equations [6].

$$E_{n-2,i} = \bigoplus_{j=0}^{n-3} E_{j,<i+j+2>_n}, \ (0 \leqslant i \leqslant n-1) \tag{4}$$

$$E_{n-1,i} = \bigoplus_{j=0}^{n-3} E_{j,<i-j-2>_n}, \ (0 \leqslant i \leqslant n-1) \tag{5}$$

According to Equation (3) and the replacement of $i := j$ and $j :=< i+j+2 >_n$, $E_{n-2,i}$ is actual $N_{n-2,i}$, while $E_{j,<i+j+2>_n}$ is actual $N_{<\frac{n-3}{2}\cdot(<i+j+2>_n-j)>_{n-2},<i+j+2>_n}$. Therefore, after reordering, Equation (4) transforms into the following equation.

$$N_{n-2,i} = \bigoplus_{j=0}^{n-3} N_{<\frac{n-3}{2}\cdot(<i+j+2>_n-j)>_{n-2},<i+j+2>_n} \tag{6}$$

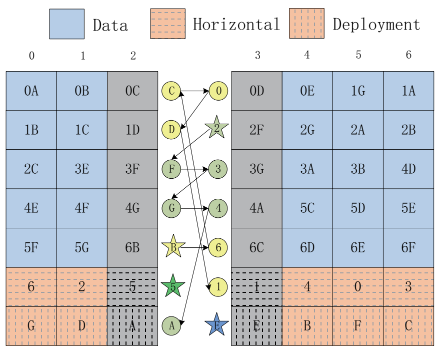


Figure 3: Recovery from disk 2 and disk 3 concurrent failures.

Similarly, after reordering, Equation (5) turns into the following equation ($i := j$ and $j :=< i-j-2 >_n$).

$$N_{n-1,i} = \bigoplus_{j=0}^{n-3} N_{<\frac{n-3}{2}\cdot(<i-j-2>_n-j)>_{n-2},<i-j-2>_n} \tag{7}$$

We can easily observe that the encoding rules of X-Code after reordering are the same as those of D-Code, i.e., Equation (6),(7) accord with Equation (1),(2) respectively. Therefore, we can conclude that D-Code can be generated by reordering the elements of each column of X-Code based on Equation (3). □

Theorem 2: D-Code can tolerate any two concurrent disk failures **when and only when n is a prime number**.

Proof: According to Theorem 1, D-Code can be generated by reordering the elements of each column of X-Code based on Equation (3). According to Lemma 2, we can deduce that D-Code has the same fault-tolerance as X-Code. Therefore, D-Code can tolerate any two concurrent disk failures when n is a prime number, because the fault tolerance of X-Code is exact 2 when and only when n equals to a prime. □

C. Reconstruction

We now discuss how D-Code recovers from disk failures. As referred above, each parity element is calculated by the XOR sum of its related data elements, thus we can recover one lost element in Equation (1) or (2) by retrieving the other $n-3$ elements and calculate the XOR sum of them. If a single disk fails, we can easily reconstruct each lost element based on Equation (1) or Equation (2), because each equation in Equation (1)(2) just contains no more than one element in each disk.

When suffers from double disk failures (suppose the two failed disks are f_1 and f_2, $0 \leqslant f_1 < f_2 \leqslant n-1$), D-Code can reconstruct all failed elements based on Theorem 2 by starting from 4 parity elements ($P_{n-2,<f_1-1>_n}$, $P_{n-2,<f_2-1>_n}$, $P_{n-2,<f_1+1>_n}$, $P_{n-2,<f_2+1>_n}$). For example, Figure 3 illustrates an example of recovery from disk 2 and disk 3 concurrent failures. As it shows, we first recover $D_{1,3}$ based on $P_{5,1}$ (P_{n-2,f_1-1}) and Equation (1) (The same number '2'), because this equation doesn't cover any element in disk 2. We then recover $D_{2,2}$ in terms of $D_{1,3}$ and Equation (2) (The same letter 'F'), and then recover the next element, and so on. The recovery sequence is $\{D_{1,3} \rightarrow D_{2,2} \rightarrow D_{2,3} \rightarrow D_{3,2} \rightarrow D_{3,3} \rightarrow P_{6,2}\}$. Similarly, we can recover $D_{4,2}$ by $P_{6,4}$ (P_{n-1,f_2+1}) and Equation (2) (The same letter 'B'), and then recover the next element, and so on. The recovery sequence is $\{D_{4,2} \rightarrow D_{4,3} \rightarrow D_{0,2} \rightarrow D_{0,3} \rightarrow D_{1,2} \rightarrow P_{5,3}\}$. Following this method, we can directly recover $P_{5,2}$ and $P_{6,2}$ based on Equation (1)(2), respectively. Now, all lost elements have been reconstructed.

D. Features

The Optimal Storage Efficiency: Based on the correctness proof, we can easily deduce that D-Code is an MDS code and thus provides the optimal storage efficiency.

The Optimal Encoding/Decoding Computational Complexity: According to Equation (1)(2), we can easily find that D-Code needs $n-3$ XOR operations to calculate each parity element, thus the total amount of XOR operations is $2n(n-3)$, because there are $2n$ parity elements in total. On the other hand, since a stripe of D-Code has $n(n-2)$ data elements, the amount of XOR operations per data element is exact $2n(n-3)/n(n-2) = 2 - 2/(n-2)$, which is the optimal encoding computational complexity in RAID-6 MDS codes [4]. On the other hand, similar as X-Code, D-Code needs to use all $2n$ equations to recover from double disk failures, thus the decoding computation complexity is $2n(n-3)/2n = (n-3)$ per failed element, which is the optimal decoding computation complexity in RAID-6 MDS codes [7].

The Optimal Update Complexity: In D-Code, each data element is used to compute two and only two parities, while the parities are independent on each other. Therefore, in order to update each data element, D-Code just needs to update exactly two additional parity elements, which is the optimal update complexity in RAID-6 codes [6].

Reducing I/O Cost to Recover from Single Failures: As referred in correctness proof, D-Code can reconstruct from disk failures by a similar recovery method as X-Code. Since Xu et. al in [18] have proved that X-Code can reduce about 25% disk reads compared to the conventional approach, our D-Code can reduce about 25% read accesses as well.

IV. I/O Loads Evaluations

In this section, we build a group of simulations to evaluate the I/O loads of D-Code by comparing with other popular RAID-6 codes under various workloads. The goal of these simulations is to demonstrate D-Code's effectiveness on both load balancing and I/O cost.

A. Evaluation Methodology

We restrict our attention to RDP-Code (over p+1 disks), H-Code (over p+1 disks), HDP code (over p-1 disks) and X-Code (over p disks), and evaluate the I/O loads under the following three representative workloads, because these workloads are commonly occurred in real storage systems and other workloads can be derived by the combination of them.

- Read-Only Workload: The Read-Only Workload is constituted by a number of read operations, where each operation can be represented as a 3-tuple $< S, L, T >$. The mathematic notation S denotes the starting element, L denotes the read length, and T denote the read times. For example, in D-Code, the tuple $< 0, 4, 5 >$ means to read 4 continuous data elements that start from $D_{0,0}$ (i.e., $D_{0,0}$, $D_{0,1}$, $D_{0,2}$, and $D_{0,3}$) 5 times. In practice, this kind of workload is usually occurred in various cloud storage systems.
- Read-Intensive Workload: The Read-Intensive Workload contains both read operations and write operations, where the ratio between reads and writes is 7 : 3. This kind of workload is usually occurred in SSD arrays due to flash's features. Similarly as read operations, each write operation also can be represented as a 3-tuple $< S, L, T >$.
- Read-Write Evenly Mixed Workload: The Read-Write Evenly Mixed Workload consists of a lot of read operations and write operations, where the ratio between reads and writes is 1 : 1. In practice, this workload usually occurred in traditional file system over disk arrays.

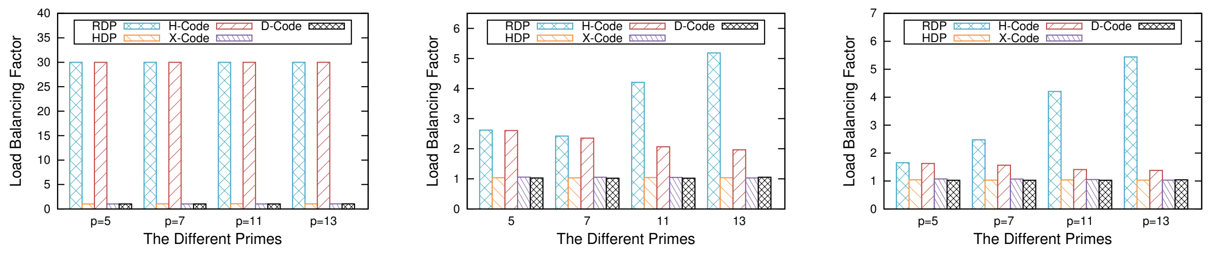
For each comparison code, we evaluate the I/O loads under above three workloads when $p = 5, 7, 11, 13$, respectively. For each workload, we randomly generate 2000 different 3-tuple $< S, L, T >$ to simulate the read/write operations, where S may be an arbitrary element of the stripe, the range of L is 1 to 20 data elements (the same range is also used in [19]), while the range of T is 1 to 1000 (the same range is used in [17] as well).

B. Evaluation Metrics

We now give the metrics for evaluating the I/O loads. Firstly, we define $L(i)$ as the number of I/O accesses in the disk i under each workload. We use L_{max} to denote the maximum number of the $L(i)$, while use L_{min} to denote the minimal number of the $L(i)$. Based on these definitions, the I/O loads can be evaluated by the following two metrics.

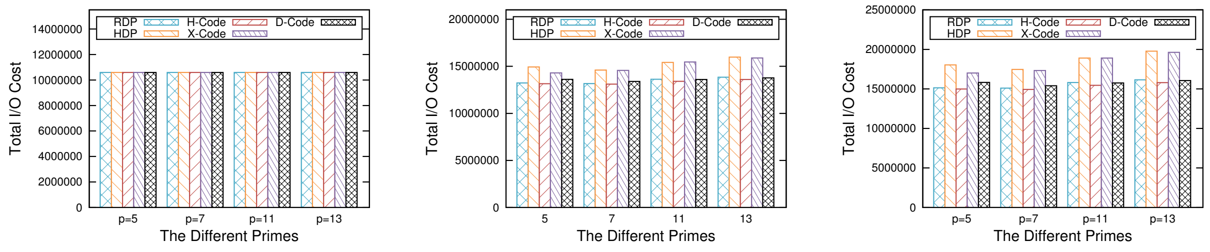
Load Balancing Factor: We define the load balancing factor LF as the ratio between L_{max} and L_{min} (which has been used in [17] as well). In mathematics, LF can be computed by the following equation.

$$LF = \frac{L_{max}}{L_{min}} \tag{8}$$



(a) Read-Only Workload. (b) Read-Intensive Workload. (c) Read-Write Evenly Mixed Workload.

Figure 4: The load balancing factor for various erasure codes under different workloads (due to the limitation of the range in y-axis, we use '30' to represent the infinity).



(a) Read-Only Workload. (b) Read-Intensive Workload. (c) Read-Write Evenly Mixed Workload.

Figure 5: The I/O cost for various codes under the different workloads.

It is easy to deduce that the smaller LF indicates the better load balancing. The lower bound of LF is 1 (i.e., $L_{max} = L_{min}$), which means that the system achieves the best load balancing.

I/O Cost: As referred in Section II-A, when process the same workload, the total number of I/O accesses in various codes are different. We define the I/O Cost ($Cost$) as the total number of I/O accesses. In mathematics,

$$Cost = \sum L(i) \tag{9}$$

We can easily deduce that the smaller $Cost$ means the lower I/O cost when processing the same workload.

C. Evaluation Results

The Load Balancing: We evaluate the load balancing factor for various codes under above three workloads, and show the result in Figure 4. As the figure shows, for read-only workload, RDP code and H-Code are bad balanced, while X-Code, HDP code and D-Code are well balanced.

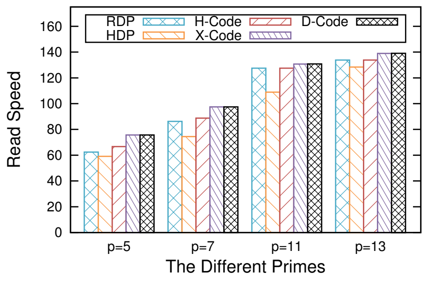
For read-intensive workload, RDP code provides bad load balancing, because the diagonal parity disk becomes the bottleneck and takes up a lot of I/O accesses. The LF in H-Code are 2.61, 2.35, 2.07 and 1.97 over $p = 5, 7, 11, 13$ respectively, which is also not well balanced. On the other hand, HDP code, X-Code and D-Code are well balanced due to the fact that the LF in them are very close to 1.

For read-write evenly mixed workload, the results show that the LF in RDP code is 1.66 to 5.44, in H-Code is 1.38 to 1.63, but in HDP code, X-Code and D-Code are only 1.03 to 1.07, which illustrate that HDP code, X-Code and D-Code are well balanced but RDP code and H-Code are not well balanced in this kind of workload.

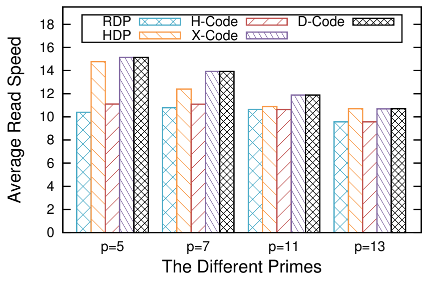
In summary, RDP code provides bad load balancing under all three workloads. H-Code provides bad load balancing under read-only workload and read-intensive workload, while achieves medium load balancing under read-write evenly mixed workload. HDP code, X-Code and our proposed D-Code achieve very good load balancing under all three workloads.

The I/O Cost: Figure 5 shows the I/O cost for various codes under different workloads. As it shows, for read-only workload, all codes provide the same I/O cost, because reads don't bring any extra disk accesses. However, when it comes to read-intensive workload and read-write evenly mixed workload, HDP code and X-Code provide much higher I/O cost than other comparison codes due to the fact that they suffer from more I/O accesses on partial stripe writes.

In statistics, when $p = 13$, D-Code provides 16.0% and 15.3% lower I/O cost than HDP code and X-Code under read-intensive workload, while achieves 23.1% and 22.2% lower I/O cost than HDP code and X-Code under read-write evenly mixed workload, respectively. Moreover, RDP code and H-Code gains a little (at most 3.4%) lower I/O Cost than our proposed D-Code due to that the total amount of disks in RDP code and H-Code ($p+1$) is larger than that of D-Code (p) caused a little better at shunting I/O accesses.



(a) Read speed.



(b) Average read speed.

Figure 6: The normal read speed for different RAID-6 codes (unit: MB/s).

In summary, D-Code provides much lower I/O cost than other well-balanced codes (HDP code and X-Code), while achieves the approximate I/O cost as not well-balanced codes (RDP code and H-Code). Based on these results, we can conclude that D-Code achieves very good I/O loads under various workloads.

V. Read Performance Experiments

In this section, we conduct a number of experiments to evaluate the read performance. We implement each code in a disk array based on Jerasure-1.2 library [11], which is an open source library and commonly used in erasure code community [20]. We select typical horizontal codes (RDP code) and typical vertical codes (HDP code, H-Code, and X-Code) as comparison, to evaluate the read performance by comparing the real speed on both normal mode and degraded mode when each code is deployed over $p = 5, 7, 11, 13$, respectively. We use both read speed and average read speed contributed from each disk as metrics, because the amount of disks in comparison codes are different and all disks in RAID system can be accessed in parallel.

A. *Experiment Environment*

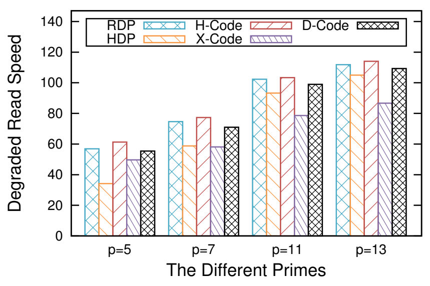
We run the experiments on a machine with Intel Xeon X5472 processor and 12 GB RAM and a disk array with 16-disks. The type of each tested disk is Seagate/Savvio 10K.3, while the model number is ST9300603SS. Each disk has 300GB capability and 10000 rmp. The operation system of the machine is SUSE with Linux fs91 3.2.16.

B. *Read Experiments in Normal Mode*

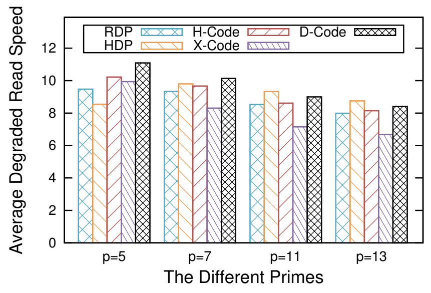
We now evaluate the read speed in normal mode and run 2000 experiments for each code over each number of prime p. For each experiment, we randomly generate the start point and the read size, where the start point may be an arbitrary data element of the stripe and the range of read size is 1 to 20 data elements. Figure 6(a) shows the read speed for various erasure codes. As it shows, D-Code and X-Code achieve very close read speed, because the data layout of them are identical. Compared to RDP code and H-Code, D-Code gains up to 21.3% and 13.5% higher read speed respectively, because there are only $p-1$ disks in RDP code that contribute to read. Though there are p disks in H-Code that contribute to read, the parities laid in middle of the stripe decrease the read speed. Moreover, compared to HDP code, D-Code owns up to 31.0% higher read speed.

On the other hand, since the disks in RAID systems can be accessed in parallel, it is rational that the more disks bring much higher read speed. Based on this reason, we calculate average read speed by computing the ratio between the read speed and the amount of disks. For example, the read speed for 8-disk RDP code (when $p = 7$) is 86.3MB/s, thus the average read speed is $\frac{86.3MB/s}{8\ disks} = 10.79$ MB/s per disk. Figure 6(b) shows the average read speed for each code. As it shows, D-Code and X-Code achieve up to 45.6% and 36.2% higher average speed than RDP code and H-Code respectively, while provide up to 12.2% higher average speed than HDP code. On the other hand, we can easily observe that when p goes large the average read speed for each code is decreased, which indicates that the read speed is not linearly increasing with the amount of disks. Therefore, though the average read speed for HDP code (over p-1 disks) seems much higher than that in RDP code and H-Code (over p+1 disks), in actual they are closely under the same number of disks. For example, the read speed for both 12-disks RDP code and 12-disks H-Code (when $p = 11$) is 127.5MB/s, while the read speed for 12-disks HDP code (when p=13) is just 128.4MB/s per disks (the read speed for 11-disks D-Code is 130.8MB/s). D-Code has less disks than RDP code and H-Code over the same $\boldsymbol{p}$, but it achieves much higher read speed, which illustrates that D-Code achieves much better read performance than these codes in normal mode.

In summary, D-Code achieves much better read performance than RDP code, H-Code and HDP code due to the higher read speed and the higher average read speed. Though X-Code achieves the similar read speed as D-Code in normal mode, it provides much lower read speed in degraded mode that we discuss in the next subsection.



(a) Degraded Read speed.



(b) Average degraded read speed.

Figure 7: The degraded read speed for different RAID-6 codes (unit: MB/s).

C. *Read Experiments in Degraded Mode*

In this subsection, we build a series of experiments to evaluate the read speed in degraded mode. Since each RAID-6 code contains k different data disk failure cases, we set 200 experiments for each possible failure case. Similar as in normal mode, for each experiment we randomly generate the start point and the read size, where the start point may be an arbitrary data element of the stripe and the range of read size is 1 to 20 data element.

Degraded read speed: Figure 7(a) shows the degraded read speed for different codes. As it shows, D-Code achieves 11.6% to 26.0% higher degraded read speed than X-Code, because it has higher possibility of continuous data elements sharing the common parities. On the other hand, D-Code provides 2.3% to 4.9% and 4.1% to 9.6% lower speed than RDP code and H-Code respectively, because the amount of disks in RDP code and H-Code is one more than that in D-Code and the horizontal parity disk helps to degraded reads (though it doesn't contribute to normal reads). Moreover, D-Code gains 4.1% to 62.4% higher degraded read speed than HDP code.

Average degraded read speed: We calculate the average degraded read speed based on the similar method referred in above subsection, and show the results in Figure 7(b). As the figure shows, D-Code achieves 5.3% to 17.1% and 3.3% to 8.5% higher average degraded read speed than RDP code and H-Code respectively. Compared to HDP code, D-Code gains 30.0% and 3.5% higher average speed when $p = 5$ and $p = 7$, but provides 3.5% and 3.9% lower average speed when $p = 11$ and $p = 13$, because D-Code has one more disk compared to HDP code and the average degraded read speed is decreased with the p increasing.

In summary, in degraded mode, D-Code, RDP code and H-Code provide high read speed, while D-Code and HDP code achieve high average speed. On the other hand, HDP code gains medium read speed, RDP code and H-Code provide medium average speed, while X-Code provides both low read speed and low average speed. All these results indicate that D-Code not only achieves good read performance in normal mode, but also works well in degraded mode.

VI. Related Work

Many RAID-6 implementations have been proposed in literature, where a typical class is to use various erasure codes, including Reed-Solomon code [22], Cauchy Reed-Solomon code [23], RDP code [4], EVENODD code [5], Liberation code [8], Liber8tion code [24], Blaum-Roth code [25], X-Code [6], HDP code[17], H-Code [7], C-Code [26], Cycle code [27], and STAIR code [28]. All these codes are MDS codes and provide the optimal storage efficiency. Some non-MDS codes such as Hover code [29], WEAVER code [30], LRC code [31], and Generalized X-Code [32]) can be used for RAID-6 implementations as well. Furthermore, some network codes like NCCloud [33] and Zigzag code [34] are other RAID-6 candidate implementations and achieve good performance on single disk failure recoveries. In this paper, we only focus on RAID-6 MDS codes, which can be further classified as horizontal codes and vertical codes.

Representative Horizontal MDS Codes for RAID-6: Reed-Solomon code is the earliest horizontal code, which is constructed over GF(2^w) under the Vandermonde matrix. Caughy RS code converts the Galois operations into the XOR operations by built the bit matrix, in order to reduce the computation complexity. Different from previous codes, EVENODD code and RDP code are specialized RAID-6 code that can tolerate up to 2 disk failures. In EVENODD code, the horizontal parities are calculated by the XOR sum of the data elements in each row, while the diagonal parities are computed by the data elements on diagonals. In RDP code, the generation of horizontal parities is the same as EVENODD code, while diagonal parities are generated by the diagonal elements including both data elements and horizontal parity elements, in order to provide the optimal encoding/decoding computation complexity.

Representative Vertical MDS Codes for RAID-6: Vertical codes usually distribute their parities among all disks, such as X-Code, HDP code and H-Code. A stripe of X-Code can be represented as a $p * p$ matrix, where p needs to be a prime number. In X-Code, the data elements are laid in the first $(p-2)$ rows, while the parity elements are deployed in the last two rows. There are two types of parities in X-Code: diagonal parities, which are generated by the XOR sum of diagonal data elements and laid in the $(p-1)th$ row, and anti-diagonal parities which are computed by anti-diagonal data elements and deployed in the pth row. HDP code uses both horizontal parities and diagonal parities, and evenly distributes all parities in the middle of stripe to optimize the load balancing and the recovery from disk failures. Unlike above codes evenly distributed the parities among all disks, H-Code stores all horizontal parities in a specialized disk and distributes diagonal parities among other disks, in order to optimize the partial stripe writes.

VII. Conclusions

In this paper, we propose a novel RAID-6 MDS code called D-Code, which uses horizontal parities and deployment parities in order to optimize the I/O loads and read performance. A stripe of D-Code is composed of an n-rows and n-columns matrix, where n needs to be a prime number. We conduct a series of simulations and experiments to evaluate our proposed D-Code, where the results show that D-Code not only provides good load balancing, but also gains lower I/O cost and better read performance than other popular RAID-6 MDS codes.

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), and the Tsinghua University Initiative Scientific Research Program.

References

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

[2] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. Mckelvie, Y. Xu, S, Srivastav, J. Wu, H. Simitci, et al. Windows azure system: a highly available cloud storage service with strong consistency. In Proceedings of ACM SOSP'11, 2011.

[3] F. J. MacWilliams and N. J. A. Sloane. The Theory of Error-Correcting Codes. New York: North-Holland, 1977.

[4] 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.

[5] M. Blaum, J. Bruck, and J. Nebib. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. IEEE Transaction on Information Theory, 45(1):46-59, Junuary 1999.

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

[7] C. Wu, S. Wan, X. He, Q. Cao and C. Xie. H-Code: A hybrid MDS array code to optimize large stripe writes in raid-6. In Proceedings of the IEEE IPDPS'11, May 2011.

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

[9] 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.

[10] D. Patterson, G. Gibson, and R. Katz. A case for redunant arrays of inexpensive disks (RAID). In Proceedings of the SIGMOD'88, 1988.

[11] 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.

[12] G. Ganger, B. Worthington, R. Hou, and Y. Patt. Disk subsystem load balancing: disk striping vs. conventional data placement. In Proceeding of the HICSS'93, 1993.

[13] A. Zomaya and Y. Teh. Observations on using genetic algorithms for dynamic load-balancing. IEEE Transactions on Parallel and Distributed Systems, 12(9):899-911, 2001.

[14] E. Bachmat, Y. Ofek, A. Zakai, M. Schreiber, V. Dubrovsky, T. Lam. and R. Michel. Load balancing on disk array storage device. US Patent No. 6711649B1, 2004.

[15] P. Scheuermann, G. Weikum, and P. Zabback. Adaptive load balancing in disk arrays. In Proceeding of the FODO'93, 1993.

[16] P. Scheuermann, G. Weikum, and P. Zabback. Data partitioning and load balancing in parallel disk systems. the VLDB Journal, 7(1):48-66, 1998.

[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'10, 2010.

[18] S. Xu, R. Li, P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. Lui. Single disk falure recovery for X-code-based parallel storage systems. IEEE Transaction on Computers, January, 2013.

[19] 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.

[20] 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, 2009.

[21] J. Schindler, S. W. Schlosser, M. Shao, A. Ailamaki, and G. R. Ganger. Atropos: A Disk Array Volume Manager for Orchestrated Use of Disks. In Proceedings of USENIX FAST'04, March, 2004.

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

[23] 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.

[24] J. Plank. A new minimun density RAID-6 code with a word size of eight. In Proceedings of the IEEE NCA'08, 2008.

[25] M. Blaum, and R. Roth. On lowest density MDS codes. IEEE Transactions on Information Theory, 45(1):46-59, 1999.

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

[27] Y. Cassuto and J. Bruck. Cyclic lowest density MDS array codes. IEEE Transaction on Information Theory, 55(4):1721-1729, 2009.

[28] 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, February 2014.

[29] J. Hafner. HoVer erasure codes for disk arrays. In Proceedings of IEEE/IFIP DSN'06, June, 2006.

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

[31] 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.

[32] X. Luo and J. Shu. Generalized X-Code: An Efficient RAID-6 Code for Arbitrary size of Disk Array. ACM Transaction on Storage, 8(3), September, 2012.

[33] Y. Hu, H. Chen, P. Lee, and Y. Tang. NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In Proceedings of USENIX FAST'12, 2012.

[34] I. Tamo, Z. Wang, and J. Bruck. Zigzag Codes: MDS Array Codes with Optimal Rebuilding. IEEE Transaction on Information Theory, 55(3), March, 2013.