HV Code: An All-Around MDS Code for RAID-6 Storage Systems

Zhirong Shen, Jiwu Shu, *Member, IEEE*, and Yingxun Fu

Abstract—The increasing expansion of data scale leads to the widespread deployment of storage systems with larger capacity and further induces the climbing probability of data loss or damage. The *Maximum Distance Separable* (MDS) code in RAID-6, which tolerates the concurrent failures of any two disks with minimal storage requirement, is one of the best candidates to enhance the data reliability. However, most of the existing works in this literature are more inclined to be specialized and cannot provide a satisfied performance under an all-round evaluation. Aiming at this problem, we propose an all-round MDS code named *Horizontal-Vertical Code* (HV Code) by taking advantage of horizontal parity and vertical parity. HV Code achieves the perfect I/O balancing and optimizes the operation of partial stripe writes, while preserving the optimal encoding/decoding/update efficiency. Moreover, it owns a shorter parity chain which grants it a more efficient recovery for single disk failure. HV Code also behaves well on degraded read operation and accelerates the reconstruction of double disk failures by executing four recovery chains in parallel. The performance evaluation demonstrates that HV Code well balances the I/O distribution. HV Code also eliminates up to 32.2 percent I/O operations for partial stripe writes in read-modify-write mode, and reduces up to 28.9 percent I/O operations for partial stripe writes in reconstruct-write mode. Moreover, HV Code reduces 5.4∼39.8 percent I/O operations per element for the single disk reconstruction, decreases 8.3∼39.0 percent I/O operations for degraded read operations, and shortens 47.4∼59.7 percent recovery time for double disk recovery.

Index Terms—Erasure codes, RAID-6, degraded read, load balancing, single failure recovery

1 Introduction

With the rapid development of cloud storage, the size of created data to be kept is amazingly expanding, resulting in the strong demand of the storage systems with larger capacity (e.g., GFS [2] and Windows Azure [3]). Increasing the storage volume is usually achieved by equipping with more disks, but it also comes with the rising probability of multiple disk failures [4], [5] with the system scales up. To provide a reliable and economical storage service with high performance, *Redundant Arrays of Inexpensive (or Independent) Disks* (RAID) receives tremendous attention and is widely adopted nowadays. Among the various branches of RAID, the *Maximum Distance Separable* (MDS) code of RAID-6 offering the tolerance for concurrent failures of any two disks with optimal storage efficiency, becomes one of the most popular solutions.

In RAID-6 system, the original data will be partitioned into many pieces with constant size (denoted as "*data elements*") and the redundant pieces with the same size (denoted as "*parity elements*") are calculated over a subgroup of data elements. Once a disk failure happens (e.g., hardware malfunction or software error), the surviving data elements and parity elements will be selectively retrieved to reconstruct the elements on the dispirited disk. In the meantime of disk failure, the storage system may always receive read operations to the data elements resided on the corrupted disk (this operation is referred as "*degraded read*"). Therefore, to timely response to user's I/O requests and improve data reliability, it is extremely critical to efficiently process data recovery and degraded read.

Meanwhile, in general, "healthy" RAID-6 storage systems also have to cope with the frequent read/write accesses to the hosted data. Compared with degraded read, we refer the read operations issued to intact storage systems as "*normal read*". For the write operation, it can be classified into *full stripe write* and *partial stripe write* according to the size of operated data. Full stripe write completely writes all the data elements in the stripe and its optimality can be promised by MDS Code [6]. In this paper, we focus on the partial stripe write that operates a subset of data elements in a stripe. It can be handled by two modes named *read-modify-write* (RMW) mode [6], [7] and *reconstruct-write* (RW) mode [8], [9], [10], [11], which may cause different size of I/O operations. Both of these two write modes require to update the associated parity elements, and thus amplify the write size. Due to above reasons, in the circumstance with intensive write requests, storage systems may easily suffer from the unbalanced load and are likely to be exhausted with the absorption of a considerable number of extra I/O operations. Thus, to accelerate the write operation and improve the system reliability, a RAID-6 storage system that can well balance the load and be efficient to cope with the partial stripe writes is an imperative need.

Based on the above concerns, we extract the following five metrics to roughly evaluate the performance of RAID-6 storage systems. 1) The capability to balance the I/O distribution; 2) The performance of partial stripe writes; 3) The

• *The authors are with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: zhirong.shen2601@gmail.com, shujw@tsinghua.edu.cn, fu-yx10@mails.tsinghua.edu.cn.*

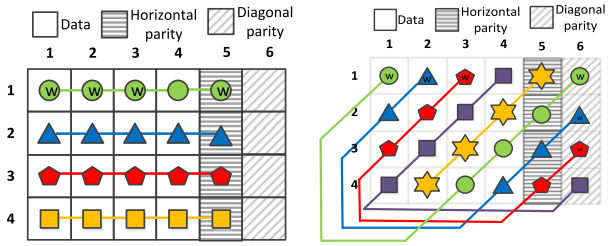
Manuscript received 22 Mar. 2015; revised 18 June 2015; accepted 22 July 2015. Date of publication 4 Aug. 2015; date of current version 18 May 2016. Recommended for acceptance by H. Jin.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2015.2464800

efficiency to reconstruct the failed disk (disks); 4) The efficiency of read operations, including normal read and degraded read; 5) The encoding/decoding/update complexity. It is extremely important to design this kind of RAID-6 code that behaves well on all the above five metrics, so that a storage system equipped with this RAID-6 code can not only efficiently cope with various I/O operations existing in real scenario, but also timely reconstruct the lost data to enhance the data reliability. In this paper, we call this kind of RAID-6 code the "*all-around*" RAID-6 code.

Deep investigations (e.g., [6], [12], [13], [14], [15], [16]) have been endeavored to pursue the optimization on one of these metrics, *yet all of them are attending to one goal and losing another*. On the aspect of balancing the I/O operations, both X-Code [12] and HDP Code [13] can evenly disperse the load to the whole disk array, but the former has a poor performance on partial stripe writes while the latter suffers high update computation complexity. On the metric of partial stripe writes and degraded read, H-Code [6] achieves excellent performance for the writes to two continuous data in read-modify-write mode, but it has the flaw on balancing the load and could not provide a satisfied repair efficiency to recover the corrupted disks. In the case of disk reconstruction, HDP-Code [13] requires less elements, but its non-optimal update complexity may introduce more I/O operations. More details about the weaknesses among existing MDS codes are discussed in Section 2.3.

In this paper, we propose a novel XOR-based MDS RAID-6 code called *Horizontal-Vertical Code* (HV Code) by taking advantage of horizontal parity and vertical parity. By evenly dispersing parity elements across the storage systems, HV Code can well balance the I/O load. It accelerates the repair process for disk (or disks) reconstruction as well by reducing the number of data elements involved in the generation of parity elements, so that less elements should be retrieved for every lost element. To optimize the efficiency of the writes to two continuous data elements, HV Code utilizes the advantage of horizontal parity which only renews the horizontal parity element once for the updated data elements in the same row, and designs a dedicate construction for the vertical parity to ensure the last data element in the ith row will share a same vertical parity element with the first data element in the $(i+1)$th row. In addition, HV Code also provides competitive performance on degraded read by utilizing the horizontal parity and still retains the optimal encoding/decoding/update computation complexity. *Our contributions* are summarized as follows:

1) We propose an all-around MDS code named HV Code, which well balances the load to the disks and offers an optimized partial stripe write experience. Meanwhile, HV Code also reduces the average recovery I/O to repair every failed element, provides efficient degraded read operations, and still retains the optimal encode/decode/update efficiency.
2) To demonstrate the efficiency of HV Code, we conduct a series of intensive experiments on the metrics of load balancing, partial stripe writes, nomal/degraded read operation, and reconstruction for the single/double disk failures. The results show that HV Code achieves the same load balancing rate as X-code and HDP Code. It significantly decreases up to 32.2 percent I/O operations for partial stripe writes in read-modify-write mode, and reduces up to 28.9 percent I/O operations for partial stripe writes in reconstruct-write mode. Moreover, it eliminates up to 39.0 percent read operations for degraded read operation. With the aspect of recovery I/O, HV Code reduces 5.4∼39.8 percent I/O operations to repair a lost element during the single disk reconstruction compared with its competitors (i.e., RDP Code, HDP Code, X-Code and H-Code). It achieves nearly the same time efficiency of X-Code in double disk recovery by decreasing 47.4∼59.7 percent recovery time compared with other three typical codes.



(a) The Horizontal Parity. (b) The Diagonal Parity.

Fig. 1. The layout of RDP code with $p+1$ disks ($p=5$). $\{E_{1,1},\ldots,E_{1,5}\}$ is a horizontal parity chain and $\{E_{1,1},E_{4,3},E_{3,4},E_{2,5},E_{1,6}\}$ is a diagonal parity chain. Their length is 5.

The rest of this paper is organized as follows. We first introduce the background knowledge of MDS codes and the motivation of this paper in Section 2, and then present the detailed design of HV Code in Section 3. The property analysis of HV Code will be given in Section 4 and a series of intensive evaluations is conducted to evaluate the performance of HV Code and other representative codes in Section 5. Finally, we conclude our work in Section 6.

2 Background and Motivation

2.1 Terms and Notations

To give a better understanding about the research background of RAID-6 codes, we first summarize the terms and notations frequently referred in this paper.

- *Data element and parity element.* Element is the basic operated unit in RAID-6 systems and can be treated as an unit of a disk, such as a byte or a sector. The *data element* contains the original data information, while the *parity element* keeps the redundant information. In Fig. 1a, $E_{1,1}$ is a data element and $E_{1,5}$ is a parity element, where $E_{i,j}$ means the element on the ith row and the jth column.
- *Stripe.* A maximal set of data elements and parity elements that have the dependent relationship connected by an erasure code. Fig. 1 shows the layout of a stripe in RDP Code, which can be treated as a $(p-1)$-row-$(p+1)$-column matrix.
- *Horizontal parity.* The horizontal parity element is calculated by performing the XOR operations among the data elements in the same row. For instance, in Fig. 1a, the horizontal parity element $E_{1,5} := \sum_{i=1}^{4} E_{1,i} := E_{1,1} \oplus \cdots \oplus E_{1,4}$.

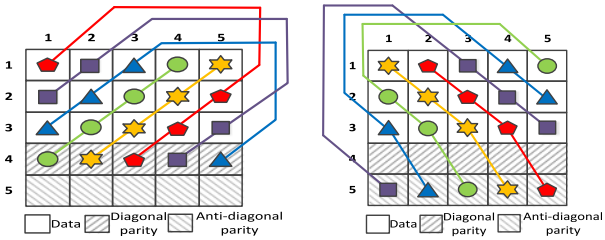
- *Diagonal parity and anti-diagonal parity.* The diagonal (resp. anti-diagonal) parity connects the elements following the diagonal (resp. anti-diagonal) line. In RDP Code [14], the horizontal parity element will participate in the calculation of diagonal parity element. For example, the diagonal parity element $E_{1,6} := E_{1,1} \oplus E_{4,3} \oplus E_{3,4} \oplus E_{2,5}$ as shown in Fig. 1b.
- *Vertical parity.* It is usually adopted by vertical codes, such as P-Code [15] and B-Code [17]. In the vertical parity calculation, the candidate data elements should be picked out first and then performed the XOR operations.
- *Parity chain and its length.* A parity chain is composed of a group of data elements and the generated parity element. For example, $E_{1,1}$ involves in two parity chains in Fig. 1, i.e., $\{E_{1,1}, E_{1,2}, \ldots, E_{1,5}\}$ for horizontal parity and $\{E_{1,1}, E_{4,3}, E_{3,4}, E_{2,5}, E_{1,6}\}$ for diagonal parity. The length of a parity chain is denoted by the number of the included elements.
- *Recovery chain.* It is constituted by a subgroup of failed elements that have dependence in double disk reconstructions. The elements in a recovery chain will be repaired in an order. For example, there are four recovery chains in Fig. 6, where $E_{2,3}$, $E_{1,1}$, $E_{1,3}$, and $E_{2,1}$ belong to the same recovery chain.
- *Continuous data elements.* In erasure coding, a data file will be partitioned into many data elements that will then be continuously and horizontally placed across disks to maximize the access parallelism [6], [13], [18]. The data elements that are logically neighboring are called "continuous data elements". For example, $E_{1,1}$ and $E_{1,2}$ in Fig. 1a are two continuous data elements, and so as $E_{1,4}$ and $E_{2,1}$.

2.2 The MDS Codes of RAID-6 Storage Systems

According to the storage efficiency, current erasure codes to realize RAID-6 function can be divided into *Maximum Distance Separable* codes and non-MDS codes. MDS codes reach optimal storage efficiency while non-MDS codes sacrifice storage efficiency to run after the improvement on other recovery metrics. The representative MDS codes for RAID-6 realization include Reed-Solomon Code [19], Cauchy Reed-Solomon Code [20], EVENODD Code [16], RDP Code [14], B-Code [17], X-Code [12], Liberation Code [21], Liber8tion Code [22], P-Code [15], HDP Code [13], and H-Code [6]. The typical non-MDS codes are Pyramid Code [23], WEAVER Code [24], Code-M [25], HoVer Code [26], Local Reconstruction Codes [3] and its application [27], and Flat XOR-Code [28]. In this paper, we mainly consider MDS codes in RAID-6, which can be classified into horizontal codes and vertical codes according to the placement of parity elements.

The horizontal codes of RAID-6 systems. Horizontal codes are well known as the first studies of RAID-6 systems. They are usually constructed over $m+2$ disks and demanded to reserve two dedicated disks to place parity elements.

As the ancestor of horizontal codes, Reed-Solomon Code is constructed over Galois field $GF(2^w)$ by employing Vandermonde matrix. Its operations (i.e., multiplication and division) are usually implemented in Galois field and this high computation complexity seriously limits its realization in practice. To mitigate this overhead, Cauchy Reed-Solomon Code introduces the binary bit matrix to convert the complex Galois field arithmetic operations into single XOR operations.



(a) The Diagonal Parity. (b) The Anti-diagonal Parity.

Fig. 2. The layout of X-code with p disks ($p = 5$).

EVENODD Code and RDP Code are the typical parity array codes. By performing the XOR operations instead of finite field arithmetic, they outperform Reed-Solomon Code on the metrics of realization and efficiency. Both of them utilize horizontal parity and diagonal parity to realize their constructions and RDP makes some differences when building the diagonal parity to achieve a better performance.

Horizontal codes own an advantage that it can be built on any number of disks, but they usually cannot approach optimal update complexity.

The vertical codes of RAID-6 systems. Rather than separating the storage of data elements and parity elements, vertical codes store them together in the disk.

X-Code [12] (as shown in Fig. 2) is construed over p disks (p is a prime number) by using both diagonal parity and anti-diagonal parity. HDP-Code [13] is proposed to balance the load in a stripe by employing horizontal-diagonal parity, in which the diagonal parity element joins the calculation of horizontal parity element. H-Code [6] optimizes the partial stripe writes to continuous data elements in read-modify-write mode. It gathers the horizontal parity elements on a dedicated disk and spreads the $(p-1)$ anti-diagonal parity elements over other p disks.

2.3 The Remained Problems of Existing MDS Codes

Though continuous efforts have been made to greatly promote the diversity and maturity of RAID-6 storage systems, most of the existing works cannot simultaneously address the following problems. These problems will potentially threaten the system reliability and degrade the system performance. Table 1 gives a brief summary of existing typical RAID-6 codes.

Load balancing. The unbalanced I/O to a storage system will not only extend the operation time, but also may make the most loaded disk easily tired out, thus degrading the data reliability [13], [29]. Aiming at this problem, the study on balancing I/O to disks has been considered for a period of time [13], [30].

The traditional method adopts "stripe rotation" (i.e., rotationally shift the parity disks among different stripes) to uniformly distribute I/O operations across all the stripes. This method only takes effect when the workload is uniform among the stripes, which actually does not accord with all the I/O distributions in the real application. In the scenario that different stripes have different access frequencies, even when "stripe rotation" is applied, the stripe hosting hotter

TABLE 1
A Brief Summary of Existing Typical RAID-6 Codes

Metrics	RDP Code [14]	HDP Code [13]	X-Code [12]	H-Code [6]	EVENODD Code [16]
Load Balancing	unbalanced	**balanced**	**balanced**	unbalanced	unbalanced
Update Complexity	$>$ extra 2 updates	3 extra updates	**2 extra updates**	**2 extra updates**	$>$ 2 extra updates
Double Disk Reconstruction	2 recovery chains	2 recovery chains	**4 recovery chains**	2 recovery chains	2 recovery chains
Parity Types[1]	HP, DP	HDP, ADP	DP, ADP	HP, ADP	HP, DP
Parity Chain Length	p	$p-2, p-1$	$p-1$	p	$p+1$

1.*"HP" denotes horizontal parity, "DP" and "ADP" represent diagonal parity and anti-diagonal parity respectively, and "HDP" is short for horizontal-diagonal parity.*

(resp. colder) data will receive more (resp. less) access requests, still causing unbalanced I/O distribution. Therefore, to well balance the load, a better method is to evenly disseminate the parity elements among the stripe.

For RDP Code [14], EVENODD Code [16], Liberation Code [21], and H-Code [6], which require dedicated disk to place parity elements, will easily cause non-uniform I/O distribution.

Partial stripe writes. The partial stripe writes to continuous data elements is a frequent operation, such as backup and virtual machine migration. As a data element is usually associated with the generation of two parity elements in RAID-6 codes, an update to a data element will also renew at least two related parity elements. This property results in various behaviors when RAID-6 codes meet partial stripe writes.

Generally, there are two modes of partial stripe write in practical applications, i.e., read-modify-write mode [6], [7] and reconstruct-write mode [9], [10], [11]. In both of these two modes, we refer the data elements to be written as "old" data elements and denote the parity elements associated with the old data elements as "old" parity elements. Similarly, we call the data (resp. parity) elements after being updated as "new" data (resp. parity) elements.

For read-modify-write mode, it needs to read both the old data elements and the old parity elements from storage devices, exclusive-OR the old data elements with old parity elements and then with the new data elements, and obtain the new parity elements. Finally, the new data elements and new parity elements will be written back. Thus the number of pre-read elements in read-modify-write mode equals the number of old data elements plus the old parity elements. Take RDP Code as an example (as shown in Fig. 3a), when the data elements $E_{1,1}$ and $E_{1,2}$ are updated in read-modify-write mode, the storage system will read these two old data elements and associated old parity elements (i.e., $E_{1,5}$, $E_{1,6}$, and $E_{2,6}$), perform the updates, and write back both the new data elements and new parity elements. Therefore, the total number of I/O operations will be 10 (i.e., five read operations and five write operations).

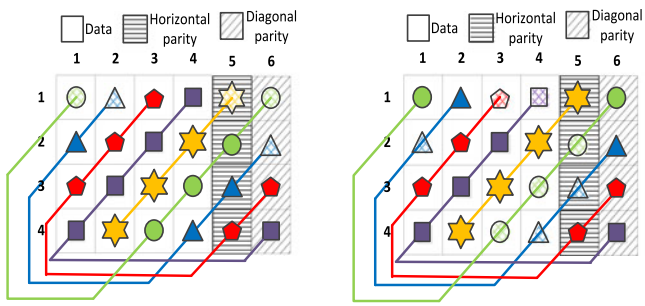
For reconstruct-write mode, it needs to read other data elements, which have the common associated old parity elements with the old data elements, recalculate the new parity elements based on the new data elements and retrieved data elements, and finally write back both the new data elements and new parity elements to the storage. Take RDP Code as an example (as shown in Fig. 3b), when $E_{1,1}$ and $E_{1,2}$ are written in reconstruct-write mode, the storage system will read the associated data elements (i.e., $E_{1,3}$, $E_{1,4}$, $E_{2,1}$, $E_{2,5}$, $E_{3,4}$, $E_{3,5}$, $E_{4,3}$, and $E_{4,4}$) that share the same old parity elements with the old elements, and write back the new data elements (i.e., $E_{1,1}$ and $E_{1,2}$) and related new parity elements (i.e., $E_{1,5}$, $E_{1,6}$, and $E_{2,6}$). The total number of I/O operations is 13 (i.e., eight read operations and five write operations).

Derived from above introduction, *for a write operation, both the read-modify-write mode and reconstruct-write mode have the same number of elements to write. Their difference lies in the elements to read and the read size.* Moreover, reconstruct-write mode is more suitable for large writes.

As a data element is usually associated with the generation of two parity elements in RAID-6 codes, an update to a data element will also renew at least two related parity elements. This property also awards horizontal parity an advantage that the update to the data elements in a row only needs to update the shared horizontal parity once.

For X-Code [12], in which any two continuous data elements do not share a common parity element, therefore the partial stripe writes will induce more extra write operations to the parity elements compared to the codes utilizing horizontal parity.

For RDP Code [14] and EVENODD Code [16], the partial stripe writes to continuous data elements will put a heavy update burden to the disk hosting diagonal parities. For example, when $E_{1,1}$, $E_{1,2}$ and $E_{1,3}$ are updated in Fig. 1, then disk #5 will only need to update $E_{1,5}$ while



(a) Read-Modify-Write Mode. (b) Reconstruct-Write Mode.

Fig. 3. An example of partial stripe writes in two different modes. Suppose $E_{1,1}$ and $E_{1,2}$ are old data elements. The non-filled shape denotes the element that is read out. (a) In read-modify-write mode, it needs to first read out the old data elements (i.e., $E_{1,1}$, $E_{1,2}$) and their associated old parity elements (i.e., $E_{1,5}$, $E_{1,6}$, and $E_{2,6}$). It then performs the exclusive-OR operations to calculates the new parity elements, and writes back both new data elements and new parity elements. (b) In reconstruct-write mode, it reads $E_{1,3}$ and $E_{1,4}$ to calculate the new parity element $E_{1,5}$ with the new data elements $E_{1,1}$ and $E_{1,2}$. It also reads $E_{2,1}$, $E_{4,4}$, and $E_{3,5}$ to compute $E_{2,6}$. Moreover, it retrieves $E_{4,3}$, $E_{3,4}$, and $E_{2,5}$ to calculate $E_{1,6}$. Finally, it writes back the new data elements (i.e., $E_{1,1}$ and $E_{1,2}$) and the new parity elements (i.e., $E_{1,5}$, $E_{1,6}$, and $E_{2,6}$).

disk #6 has to update $E_{1,6}$, $E_{2,6}$ and $E_{3,6}$, respectively. This unbalanced I/O distribution will easily delay the write operation and even threaten the system reliability by making some disks tired out.

For HDP Code [13], its high update complexity will induce considerable I/O operations. Moreover, it does not consider the optimization in the case of continuous writes across rows.

As a delicate design, H-Code [6] is proved that it behaves well to cope with the partial stripe writes in read-modify-write mode. However, it still leaves a dedicated disk to keep horizontal parity elements. This design could cause unbalanced I/O distribution for partial stripe writes in reconstruct-write mode.

Recovery cost for disk failure. The recovery of RAID-6 systems can be classified into the single disk failure recovery [31] and the reconstruction of double disk failures [13].

For single disk failure recovery, a general way firstly proposed by Xiang et al. [32] is repair the invalid elements by mixing two kinds of parity chains subjecting to the maximum overlapped elements to be retrieved, so as to achieve the minimum recovery I/O. For example, suppose the disk #1 is disabled in Fig. 1. To repair $E_{1,1}$ and $E_{2,1}$, purely using either horizontal parity chains or diagonal parity chains will need to retrieve eight elements in total. However, if we repair $E_{1,1}$ by using horizontal parity chain and recover $E_{2,1}$ by using diagonal parity chain, such reconstruction method can make $E_{1,2}$ overlapped and we can complete the recovery by reading just seven elements. The reconstruction I/O of this method is greatly reduced but still relates to the length of parity chain. Due to the long parity chain in the existing codes, there is still room for reconstruction efficiency improvement.

Different from the selective retrieval in single disk recovery, double disk recoveries demand to fetch all the elements in the survived disks. Though different codes have various layouts, the time to read all the remained elements into the main memory is the same if the parallel read is applied. Moreover, by utilizing the parallel technology, the failed elements locating at different recovery chains can be simultaneously reconstructed. For example, in Fig. 6, the element $E_{6,3}$ and $E_{5,3}$ that are not in the same recovery chain can be rebuilt at the same time. Therefore, the parallelism of recovery chains is critical in double disk reconstructions. Among the existing MDS array codes, all of RDP Code [14], HDP Code [13], H-Code [6] and P-Code [15] can construct two recovery chains to repair the double failed disks in parallel.

Read efficiency. The read efficiency is usually seriously considered for the storage systems with intensive read requests. We concern normal read and degraded read in this paper. Normal read retrieves data elements from intact storage systems, while degraded read occurs when retrieving data elements from the storage systems with corrupted disks. Their difference is that the degraded read may retrieve additional elements for data recovery when the requested data elements happen to reside on the corrupted disks. On the contrary, the normal read only retrieves the requested data elements and will not cause extra I/O operations.

For example, suppose the disk #1 fails in Fig. 1, and the elements $E_{1,1}$, $E_{1,2}$, and $E_{1,3}$ are requested at that time, then extra elements $E_{1,4}$ and $E_{1,5}$ will be also attached to re-calculate the failed element $E_{1,1}$ by the horizontal parity chain. This example also reveals that the horizontal parity owns an advantage to improve the degraded read efficiency, because some of the requested elements (e.g., $E_{1,2}$ and $E_{1,3}$) may involve in the re-calculation of the corrupted element (e.g., $E_{1,1}$) with great probability.

For X-Code [12] and P-Code [15], based on diagonal/anti-diagonal parity and vertical parity respectively, behave a bit inefficient on degraded read compared to the codes constructed on horizontal parity.

Another influence factor to the degraded read performance is the length of parity chain. Longer parity chain probably incurs more unplanned read elements. Derived from this excuse, EVENODD Code [16], RDP Code [14], H-Code [6], and HDP Code [13] could introduce a considerable number of additional I/O operations.

TABLE 2
The Frequently Used Symbols

Symbols	Descriptions
p	a prime number to configure the stripe
$\langle\rangle$, $\langle i\rangle_p$	modular arithmetic, $i \bmod p$
$E_{i,j}$	element at the i-th row and j-th column
$\sum\{E_{i,j}\}$	sum of XOR operation among the elements $\{E_{i,j}\}$
L	length of continuous data elements to write
$\langle\frac{i}{j}\rangle_p$	if $k := \langle\frac{i}{j}\rangle_p$, then $\langle k \cdot j\rangle_p = \langle i\rangle_p$

3 The Design of HV Code

To simultaneously address the above remaining limitations, an all-around RAID-6 MDS array code should satisfy the following conditions: 1) be expert in balancing the load; 2) optimize the performance of partial stripe writes in both read-modify-write mode and reconstruct-write mode; 3) be efficient to deal with single (resp. double) disk failure (resp. failures); 4) have a good performance on both normal read and degraded read; 5) retain the optimal properties, such as encoding/decoding/update efficiency.

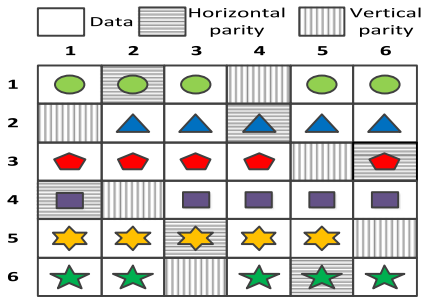
To this end, we propose an MDS code named HV Code, which makes use of horizontal parity and vertical parity and can be constructed over $(p-1)$ disks (p is a prime number). Before presenting the construction of HV Code, we first list the frequently used symbols in Table 2.

3.1 Data/Parity Layout and Encoding of HV Code

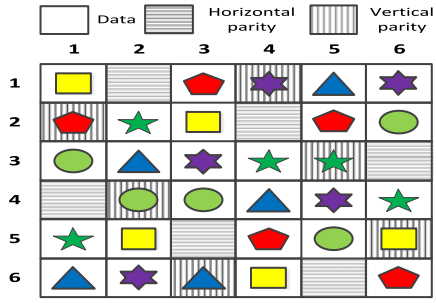
A stripe of HV Code can be represented by a $(p-1)$-row-$(p-1)$-csolumn matrix with a total number of $(p-1)\times(p-1)$ elements. There are three kinds of elements in the matrix: *data elements*, *horizontal parity elements*, and *vertical parity elements*. Suppose $E_{i,j}(1 \le i, j \le p-1)$ denotes the element at the ith row and jth column. In HV Code, the horizontal parity elements and the vertical parity elements are calculated by the following equations.

Horizontal parity element encoding:

$$E_{i,\langle 2i\rangle_p} := \sum_{j=1}^{p-1} E_{i,j} \quad (j \neq \langle 2i\rangle_p, j \neq \langle 4i\rangle_p). \tag{1}$$



(a) Horizontal Parity Encoding: a horizontal parity element can be calculated by XOR operations among the data elements in the same row. For example, $E_{1,2} := E_{1,1} \oplus E_{1,3} \oplus E_{1,5} \oplus E_{1,6}$.



(b) Vertical Parity Encoding: a vertical parity element $E_{i,j}$ can be calculated by XOR operations on selected data elements $\{E_{i',j'}\}$ satisfying the condition: $\langle 2i' + j \rangle_p = j'$. For example, $E_{1,4} := E_{6,2} \oplus E_{3,3} \oplus E_{4,5} \oplus E_{1,6}$.

Fig. 4. The layout of HV Code with $(p-1)$ disks ($p = 7$).

Vertical parity element encoding:

$$E_{i,\langle 4i \rangle_p} := \sum_{j=1}^{p-1} E_{k,j} \quad (j \neq \langle 8i \rangle_p, j \neq \langle 4i \rangle_p). \tag{2}$$

k, j, i should satisfy the condition: $\langle 2k + \langle 4i \rangle_p \rangle_p = j$. This expression can also be simplified as $\langle 2k + 4i \rangle_p = j$. Then we can obtain k according to the following equations:

$$k := \left\langle \frac{j-4i}{2} \right\rangle_p := \begin{cases} \frac{1}{2}\langle j-4i \rangle_p & (\langle j-4i \rangle_p = 2t), \\ \frac{1}{2}(\langle j-4i \rangle_p + p) & (\langle j-4i \rangle_p = 2t+1). \end{cases}$$

Notice that if u satisfies the condition $\langle u \cdot j \rangle_p = \langle i \rangle_p$, then we express u as $u := \langle \frac{i}{j} \rangle_p$. Fig. 4 shows the layout of HV Code for a six-disk system ($p = 7$). A horizontal parity element (represented in horizontal shadow) and a vertical parity element (represented in vertical shadow) are labeled in every row and every column.

Fig. 4a illustrates the process of encoding the horizontal parity elements. By following Equation (1), the horizontal parity elements can be calculated by simply performing modular arithmetic and XOR operations on the data elements with the same shape. For example, the horizontal parity element $E_{1,2}$ (the row id $i = 1$) can be calculated by $E_{1,1} \oplus E_{1,3} \oplus E_{1,5} \oplus E_{1,6}$. The vertical parity element $E_{1,4}$ ($i = 1$) should not be involved in the encoding of $E_{1,2}$, because $E_{1,4}$ is at the $\langle 4i \rangle_p$th column.

Fig. 4b shows the process of encoding a vertical parity element. Every vertical parity element is calculated by the data elements with the same shape according to Equation (2). For example, to calculate the vertical parity element $E_{1,4}$ (the row id $i = 1$), we should first pick out the involved data elements $\{E_{k,j}\}$ based on Eq. (2). When $j = 1$, then $j = \langle 8i \rangle_p$, which violates the requirements in Eq. (2). When $j = 2$, then $k := \langle \frac{j-4i}{2} \rangle_p := \langle -1 \rangle_p := 6$ and $E_{6,2}$ is positioned. By tracking this path, the following data elements (i.e., $E_{3,3}$, $E_{4,5}$, and $E_{1,6}$) are then fetched. Second, by performing XOR operations among these data elements, the vertical parity element $E_{1,4}$ will be computed as $E_{1,4} := E_{6,2} \oplus E_{3,3} \oplus E_{4,5} \oplus E_{1,6}$.

3.2 Construction Process

Based on the layout and encoding principle, we take the following steps to construct HV Code.

1) partition the disk according to the layout of HV Code and label the data elements in each disk;
2) encode the horizontal parity elements and the vertical parity elements respectively according to Equation (1) and Equation (2).

3.3 Proof of Correctness

The detailed proof can be referred in Appendix, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TPDS.2015.2464800.

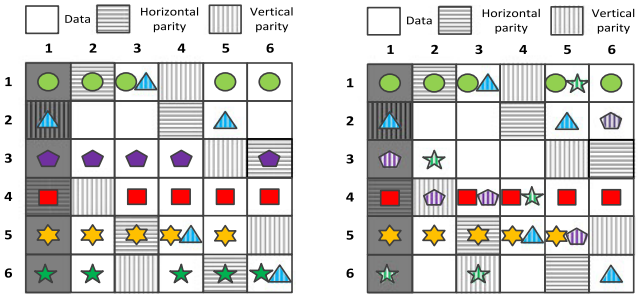
3.4 Reconstruction

In this section, we will discuss how to perform the reconstruction when a failure happens. As a RAID-6 code, HV Code can tolerate up to any two disk failures. Here, we mainly consider three basic kinds of failures: *the failure of an element*, *the failure of a single disk*, and *concurrent failures of double disk*. Other possible failure cases tolerated by RAID-6 systems (e.g., multiple element failures in a disk) can be covered by the combination of these three basic failures.

The recovery of an element. There are two possibilities when an element fails, resulting in different reconstruction methods. If the failed element is a parity element, the recovery can follow either Equation (1) or Equation (2) by using the related $(p-3)$ data elements. In the case when a data element (suppose $E_{i,j}$) is broken, it can be recovered by using either horizontal parity chain or vertical parity chain. When repairing $E_{i,j}$ by horizontal parity chain, the horizontal parity element $E_{i,\langle 2i \rangle_p}$ and other related $(p-4)$ data elements should be first retrieved, then $E_{i,j}$ can be repaired as:

$$E_{i,j} := \sum_{k=1}^{p-1} E_{i,k} \quad k \neq j, k \neq \langle 4i \rangle_p. \tag{3}$$

Similarly, when reconstructing $E_{i,j}$ by the vertical parity chain, the vertical parity element $E_{s,\langle 4s \rangle_p}$ can be obtained first, where $\langle 4s \rangle_p := \langle j - 2i \rangle_p$. Then



(a) A non-optimized scheme. (b) An optimized scheme.

Fig. 5. Two recovery schemes to repair disk #1 in HV Code ($p = 7$). The filled pattern elements participate in horizontal parity chain while the stripy elements involve in the vertical parity chain. (a) A non-optimized scheme needs to read 21 elements from surviving disks. (b) The optimized scheme just needs to read 18 elements for recovery.

$$E_{i,j} := \sum_{l=1}^{p-1} E_{k,l} \oplus E_{s,\langle 4s\rangle_p}, \tag{4}$$

where $l \neq j$, $l \neq \langle 4s\rangle_p$, $l \neq \langle 8s\rangle_p$, and $\langle 2k + 4s\rangle_p = l$.

The recovery of a single disk failure. There are two methods to recover a single failed disk. First, we can separately repair the failure elements, either data elements and parity elements, by directly using Equations (1), (2), (3), (4) without considering their dependence. However, this method may cause considerable number of elements to be retrieved.

Before introducing the second method, we first define the concept of *"recovery equation"*. A recovery equation is a collection of elements in a stripe that whose XOR sums equal zero. We take the three recovery equations that $E_{1,1}$ joins as an example.

- $\{E_{1,1}, E_{1,2}, E_{1,3}, E_{1,5}, E_{1,6}\}$ (i.e., horizontal parity chain),
- $\{E_{1,1}, E_{5,2}, E_{2,3}, E_{6,4}, E_{5,6}\}$ (i.e., vertical parity chain),
- $\{E_{1,1}, E_{1,2}, E_{1,5}, E_{1,6}, E_{2,1}, E_{5,4}, E_{2,5}, E_{6,6}\}$.

Intuitively, if an element joins r recovery equations, then it can be repaired in r different ways just by reading other elements that in the same recovery equation and performing XOR sums among them. For example, we can repair $E_{1,1}$ by reading elements in the first recovery equation and obtain $E_{1,1} = E_{1,2} \oplus E_{1,3} \oplus E_{1,5} \oplus E_{1,6}$ Therefore, to produce the minimal number of elements to be retrieved during single disk failure recovery, one commonly adopted method is to repair each lost element by selecting an appropriate recovery equation, so that the combination of $p - 1$ chosen recovery equations produces the most elements that are overlapped. Several existing studies [18], [33] have been made to optimize the single disk failure reconstruction by reducing the number of elements to read. These methods also have effect on HV Code. Fig. 5 illustrates two recovery schemes when disk 1 fails. The non-optimized scheme in Fig. 5 requires to read 21 elements, while the optimized scheme only needs 18 elements.

The recovery of double disk failures. If double disk failures occur (suppose the corrupted disks are f_1 and f_2, where $1 \leq f_1 < f_2 \leq p - 1$), the recovery process can follow the procedures in Theorem 1 (as shown in Appendix, available in the online supplemental material). We present the detailed steps to repair double failed disks in Algorithm 1.

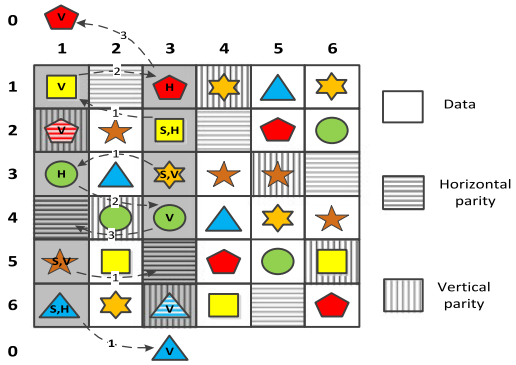


Fig. 6. An example to recover disk #1 and disk #3. The elements labeling "SH" and "SV" indicate **S**tart elements recovered by **H**orizontal parity chains, and the **S**tart elements reconstructed by **V**ertical parity chains, respectively. The element labeling "H" (resp, "V") indicates it is recovered through **H**orizontal (resp. **V**ertical) parity chain. The arrow line assigned with number denotes the recovery direction and its recovery order. There are four recovery chains, such as $\{E_{5,1}, E_{5,3}\}$ and $\{E_{3,3}, E_{3,1}, E_{4,3}, E_{4,1}\}$.

To trigger the recovery of double failed disks, we should first find four start elements by using horizontal parity chains and vertical parity chains, respectively. After that, for each start element, we can build a recovery chain by alternatively switching the parity chains. We present an example of double disk failure recovery of HV Code ($p = 7$) in Fig. 6. In this example, $E_{2,3}$, $E_{3,3}$, $E_{5,1}$, and $E_{6,1}$ are the four start elements, and $\{E_{5,1}, E_{5,3}\}$ is the recovery chain led by the start element $E_{5,1}$.

Algorithm 1. The Procedures to Recover Two Concurrent Failed Disks in HV Code

1. locate the failed disks f_1 and f_2 ($1 \leq f_1 < f_2 \leq p - 1$);
2. recover the four start elements first: $(\langle\frac{f_2 - f_1}{2}\rangle_p, f_2)$, $(\langle\frac{f_1 - f_2}{2}\rangle_p, f_1)$, $(\langle\frac{f_1}{4}\rangle_p, f_2)$, and $(\langle\frac{f_2}{4}\rangle_p, f_1)$.
3. reconstruct other missing elements by alternately shifting between f_1 and f_2.

Case 1: *start with the tuple* $(\langle\frac{f_2}{4}\rangle_p, f_1)$ *in disk* f_1, **do**
re-build elements in disk f_2 utilizing vertical parity chain and repair elements in disk f_1 using horizontal parity chain; **until** reach a parity element.

Case 2: *start with the tuple* $(\langle\frac{f_1}{4}\rangle_p, f_2)$ *in disk* f_2, **do**
re-build elements in disk f_1 utilizing vertical parity chain and repair elements in disk f_2 by horizontal parity chain; **until** reach a parity element.

Case 3: *start with the tuple* $(\langle\frac{f_1 - f_2}{2}\rangle_p, f_1)$ *in disk* f_1, **do**
re-build elements in disk f_2 utilizing horizontal parity chain and repair elements in disk f_1 using vertical parity chain; **until** reach a parity element.

Case 4: *start with the tuple* $(\langle\frac{f_2 - f_1}{2}\rangle_p, f_2)$ *in disk* f_2, **do**
re-build elements in disk f_1 utilizing horizontal parity chain and repair elements in disk f_2 using vertical parity chain; **until** reach a parity element.

4 Property Analysis

4.1 Optimal Storage Efficiency

As proved above, HV Code is an MDS code and has optimal storage efficiency [14], [19].

4.2 Optimal Construction/Reconstruction/Update Computational Complexity

For a code with m-row-by-n-column and x data elements, P-Code [15] has deduced the optimal XOR operations to each data element in construction is $\frac{3x-m\cdot n}{x}$ and the minimal XOR operations to each lost element in reconstruction is $\frac{3x-m\cdot n}{m\cdot n-x}$ [15].

Therefore, for a stripe of a code with $(p-1)$-row-by-$(p-1)$-column and $(p-3)(p-1)$ data elements, the optimal complexity of construction should be $\frac{2(p-4)}{(p-3)}$ XOR operations per data element and the optimal complexity of reconstruction should be $(p-4)$ XOR operations per lost element in theorem.

For HV Code, there are $2(p-1)$ parity elements needed to build in the construction. Each of them is calculated by performing $(p-4)$ XOR operations among the participated $(p-3)$ data elements. Therefore, the total XOR operations to generate the parity elements in a stripe is $2(p-4)(p-1)$, and the averaged XOR operations per data element is $\frac{2(p-4)}{(p-3)}$, being consistent with the above deduced result. Moreover, to reconstruct the two corrupted disks, every invalid element is recovered by a chain consisting of $(p-3)$ elements, among which $(p-4)$ XOR operations are performed. The complexity equals the optimal reconstruction complexity deduced above.

For update efficiency, every data element in HV Code joins the computation of only two parity elements, indicating the update of any data element will only renew related two parity elements. Thus, like X-Code [17] and H-Code [6], HV Code also reaches the optimal update complexity.

4.3 Achievement of Effective Load Balancing

Rather than concentrating parity elements on dedicated disks [6], [14], HV Code distributes the parity elements evenly among all the disks, which is effective to disperse the load to disks.

4.4 Fast Recovery for Disk Failure

The length of each parity chain in HV Code is $p-2$, which is shorter than those of many typical MDS codes [6], [12], [13], [14], [16] in RAID-6. The shortened parity chain decreases the number of needed elements when repairing a lost element. In addition, every disk holds two parity elements in HV Code, enabling the execution of four recovery chains in parallel in double disk repairs.

4.5 Optimized Partial Stripe Write Performance

We first dissect the performance for the writes to two continuous data elements in read-modify-write mode. The first case is when the two data elements reside in the same row, then updating them will incur only one write operation to the horizontal parity element and two separate renewals to their vertical parity elements. The second is when the renewed two data elements are in different rows, i.e., the last data element in the ith row and the first data element in the $(i+1)$th row. As described above, $E_{i,j}$ will participate in the generation of the vertical parity element residing on the $\langle j-2i\rangle_p$th disk. This rule makes $E_{i,p-1}$ and $E_{i+1,1}$, if both of them are data elements, belong to the same vertical parity chain. Therefore, the second case only needs to update a shared vertical parity element and the two corresponding horizontal parity elements. Since the HV Code's layout defines that a column will include two parity elements, there will be at least $(p-6)$ pairs[1] of two continuous data elements which locate in different rows but share the same vertical parity elements.

The proof in [6] has shown that any two data element updates should renew at least three parity elements in a stripe of a lowest density MDS code. Thus, HV Code achieves *near optimal performance* of partial stripe writes to two continuous data elements in read-modify-write mode.

For reconstruct-write mode, as discussed before, it writes back the same number of elements as in read-modify-write mode. Therefore, HV Code also optimizes the writes in reconstruct-write mode. Meanwhile, HV Code owns horizontal parity chains, which will be effective to reduce the number of read operations in reconstruct-write mode, when compared with the code that does not have horizontal parity chain, such as X-Code. This is because the updated data elements in the same row share a common horizontal parity element, and thus the elements to read for parity generation will be decreased. Moveover, the shorter parity chain in HV Code is also helpful to reduce the read operations in reconstruct-write mode, as the number of data elements that share the same parity elements with the new data elements will decrease if the parity chain is shorter.

4.6 Efficient Degraded Read

For degraded read, since HV Code has horizontal parity chains, the number of extra derived read operations will be remarkably decreased. This is because the continuous requested data elements in degraded read may share a same horizontal parity element. Therefore, some retrieved elements will be reused to join the recovery of corrupted data elements.

Let us consider the codes, which own the horizontal parity chain with the length of H, then the number of data elements in a row is $H-1$. Suppose there is a request reading n data elements in a row ($n \leq H-1$), and there happens to be a loss data element. If this corrupted data element is planned to recover by using the horizontal parity chain it joins, then the total number of elements to read is $H-2$ and the number of extra elements caused by this degraded read operation is $H-2-n$. This analysis also indicates that the code with shorter parity chain will cause less extra element retrievals for degraded read. Since the parity chain length of HV code is $p-2$, which is shorter than many well-known RAID-6 codes, such as RDP Code and X-Code, it has advantage to handle the degraded read operation.

5 Performance Evaluation

In this section, we will evaluate the performance of HV Code in terms of recovery, partial stripe writes, normal read, degraded read, and load balancing. As we mainly consider the RAID-6 setting that can tolerate any two disk failures with optimal storage efficiency, therefore we dismiss

1. There are altogether $(p-2)$ pairs of continuous two data elements that are in the different rows. So the rate is $\frac{p-6}{p-2}$, which approaches to 1 when p grows.

non-MDS codes and choose several state-of-the-art RAID-6 codes in the comparison. Specifically, we select RDP Code (over $p+1$ disks), HDP Code (over $p-1$ disks), H-Code (over $p+1$ disks), and X-Code (over p disks) to serve as the references.

Evaluation environment. The performance evaluation is run on a Linux server with a X5472 processor and 12 GB memory. The operating system is SUSE Linux Enterprise Server and the filesystem is EXT3. The deployed disk array consists of 16 Seagate/Savvio 10K.3 SAS disks, each of which owns 300 GB capability and 10,000 rmp. The machine and disk array are connected with a Fiber cable with the bandwidth of 800 MB. The five RAID-6 codes are realized based on Jerasure 1.2[34] that is widely used for erasure coding.

Evaluation preparation. We first create a file, partition it into many data elements, and encode them by using every evaluated code. Like previous works [1], [18], [35], the element size is set as 16 MB. The data elements and the encoded parity elements will then be dispersed over the disk arrays by following the layout of each code (like Fig. 1 for RDP Code and Fig. 2 for X-Code). The encoded files will be used in the next tests, such as the efficiency evaluation for partial stripe write and degraded read.

5.1 Partial Stripe Writes Efficiency

In this test, we mainly evaluate the efficiency when performing partial stripe writes in read-modify-write mode and reconstruct-write mode, respectively. In these two modes, for a write operation with the length L, both the L continuous data elements whose size is $(16 \times L)$ MB and the associated parity elements will be totally written.

To evaluate the patterns of partial stripe writes, the following three traces are mainly considered.

- *Uniform write trace*: Every access pattern simulates the operation to write a pre-defined number of continuous data elements starting from a uniformly chosen data element.
- *Random write trace*: Every access pattern (S, L, F) contains three random values, i.e., the start element S for the writes, the random length L, and the random write frequency F. For the pattern (S, L, F), it means the write operation that starts from the Sth data element and terminates at the $(S+L-1)$th data element will be executed for F times.
- *Zipf write trace*: This trace issues the write requests, where both the start element to write and the write length follow the Zipf distribution with an $\alpha = 0.9$ (like in [36]). Each Zipf trace includes 200 write patterns.

In this test, we select three uniform write traces named "uniform_w_3", "uniform_w_10", and "uniform_w_30". For uniform write trace named "uniform_w_L" (L:=3, 10, 30), we specify 1,000 write requests with the constant length L and the uniformly selected beginning. These three traces ensure the same number of data elements in a stripe is written for each code and the write frequency of each data element will be almost the same with large probability. With respect to the random write trace, we generate the patterns of (S, L, F) by employing the random integer generator [37] and the generated trace is shown in Table 3. For example, (28,34,66) means the write operation will start from the 28th data element and the 34 continuous data elements will be written for 66 times. Meanwhile, in real storage systems, the number of disks in a stripe is usually not very large. For example, a stripe in HDFS-RAID [38] includes 14 disks and a stripe in Google ColossusFS [39] has 9 disks. In the evaluations of partial stripe writes and read operations, we select $p = 13$ for each tested code, which we believe conforms to the setting of real storage systems.

TABLE 3
The Random Read/Write Pattern of (S, L, F)

(28,34,66)	(34,22,69)	(4,45,3)	(30,18,64)	(24,32,70)
(29,26,48)	(6,3,51)	(34,42,50)	(37,9,1)	(34,38,93)
(6,44,75)	(10,44,2)	(34,15,43)	(2,6,49)	(28,17,57)
(20,33,39)	(48,28,27)	(48,13,30)	(40,2,32)	(16,24,7)
(19,4,77)	(22,14,31)	(49,31,82)	(35,26,1)	(31,1,48)

Evaluation method. For the file encoded by each code, we replay the five write traces, including "uniform_w_3", "uniform_w_10", "uniform_w_30", the generated random trace, and the write trace that follows Zipf distribution respectively. During the evaluation, the following three metrics are measured.

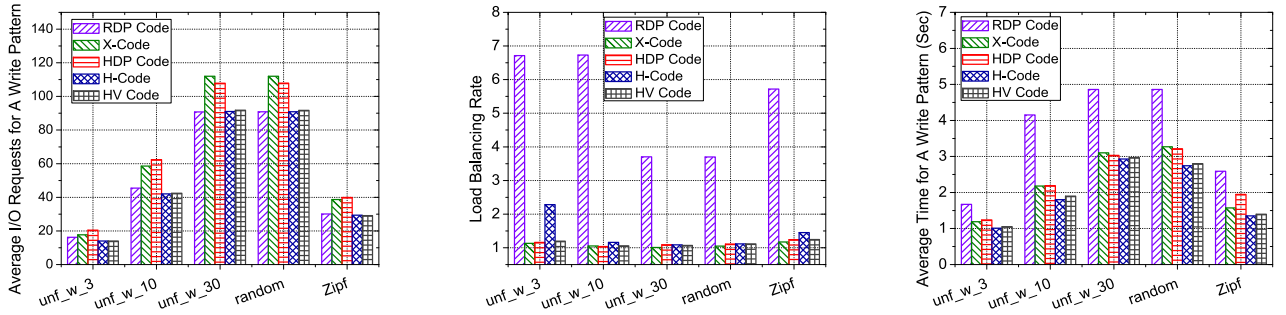
1) The total number of produced I/O operations for each write pattern. We run the five traces and record the total number of derived I/O operations (including read operations and write operations) when replaying the write trace.
2) The I/O balancing capability of each code. For the file encoded by each code, we run every trace and collect the incurred I/O operations loaded on each disk. Suppose the number of write requests arriving at the ith disk is R_i and the number of disks in a stripe is N, we can calculate the *load balancing rate* λ as the ratio of the maximum I/O loaded on a disk and the average number of I/O operations served per disk

$$\lambda := \frac{\mathrm{Max}\{R_i | 1 \le i \le N\}}{\sum_{i=1}^{N} R_i / N}. \quad (5)$$

   The smaller of λ usually indicates the better behavior of balancing the load to a stripe. We finally calculate the load balancing rate of every trace for each code.
3) The averaged time of a write pattern. For the file encoded by each code, we measure the averaged time to execute a write pattern in every trace, i.e., from the time to start the write pattern to the time when the data elements in the pattern and corresponding parity elements are completely written.

5.1.1 Read-Modify-Write Mode

Fig. 7 presents the evaluation results when running these five traces in read-modify-write mode. Fig. 7a indicates the total write operations of each code scale up with the increase of L for the uniform write traces "uniform_w_3", "uniform_w_10" and "uniform_w_30". Though X-Code retains the optimal update complexity, its construction based on diagonal parity and anti-diagonal parity engenders more write operations to the associated parity elements. Because



(a) Average I/O Operations Per Write Pattern.

(b) Load Balancing Rate.

(c) Time to Complete A Write Pattern.

Fig. 7. Partial stripe write efficiency in read-modify-write (RMW) mode ($p = 13$).

of the non-optimal update complexity, HDP Code triggers more write operations when compared to both of HV Code and H-Code, both of which reach optimal update complexity. Since both HV Code and H-Code optimize the write operation to two continuous data elements, both of them outperform other contrastive codes when conducting the continuous writes. For uniform write trace "uniform_w_10", HV Code reduces up to 27.6 and 32.2 percent I/O operations when compared to X-Code and HDP Code, respectively. When conducting the random write trace, HV Code also eliminates about 19.1 and 15.1 percent I/O operations when compared with X-Code and HDP Code, respectively. Even compared with H-Code, which has optimized partial stripe writes in read-modify-write mode, HV Code only increases marginal extra overhead, about 1.0 percent under the random write trace. Under the Zipf trace, HV Code eliminates about 24.6 and 27.0 percent I/O operations when compared with X-Code and HDP Code respectively.

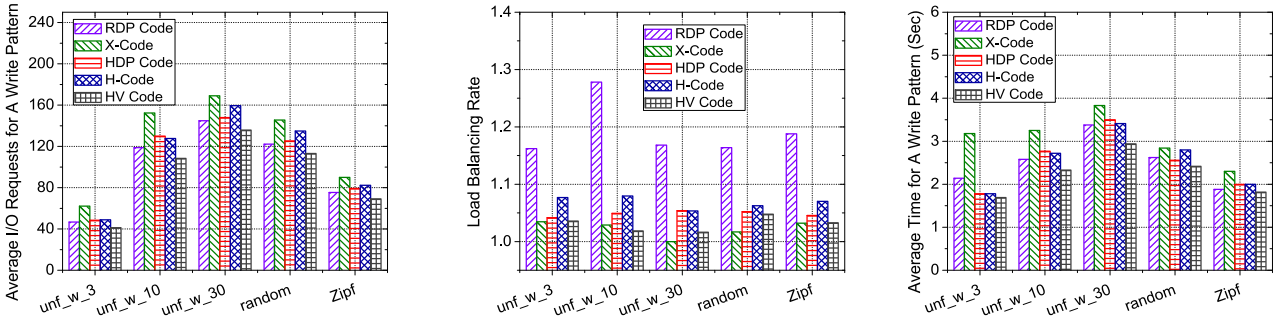
Fig. 7b illustrates the load balancing rate of various codes under different traces. Being evaluated by the three kinds of traces, RDP Code easily concentrates the writes to the parity disks and thus holds the largest load balancing rate. The load balance rates of RDP under the valuations of "uniform_w_10" and "random write trace" are 6.7 and 3.7, respectively. Though H-Code disperses the diagonal parity to the data disks, its uneven distribution of diagonal parity elements and the dedicated disk for horizontal parity storage still make it hard to achieve perfect load balancing especially when the write length is small. Its load balancing rates are 2.3 and 1.4 under the evaluations of "uniform_w_3" and "Zipf write trace", respectively. Owing to the even distribution of parity elements, HV Code, HDP Code and X-Code approach the perfect load balancing rate (i.e., 1).

The averaged time to complete a write pattern in every trace is recoded in Fig. 7c. In the figure, RDP Code needs the most time to complete the absorption of the write operations to the diagonal parity elements. The incompatible layout of X-Code and the non-optimal update complexity of HDP Code also easily extend the completion time. When performing the uniform trace "uniform_w_10", the operation time in HV Code decreases about 12.8∼54.2 percent when compared to those of RDP Code, HDP Code, and X-Code. Being evaluated by the Zipf write trace, the averaged time to complete a write pattern in HV Code is about 10.8∼45.9 percent less than those of RDP Code, X-Code, and HDP Code. However, H-Code outperforms HV Code by reducing 3.8 percent write time on this metric. This is because the number of participating disks in H-Code is larger than that of HV Code, making H-Code better at shunting the write I/O operations. This comparison also reveals the "tradeoff" brought by the shorter parity chain, which keeps down the recovery I/O for a single disk repair but is weaker at reducing the amount of the average requests on each disk.

5.1.2 Reconstruct-Write Mode

Fig. 8 shows the evaluation results when performing the five write traces in reconstruct-write mode.

Fig. 8a first illustrates the average number of I/O operations derived by a write pattern. We can find that X-Code

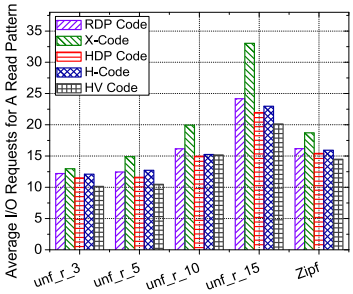


(a) Average I/O Operations Per Write Pattern.

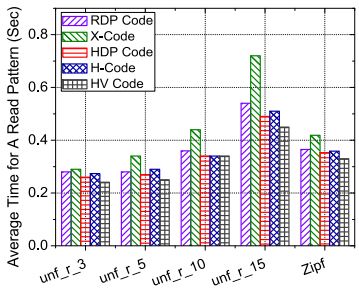
(b) Load Balancing Rate.

(c) Time to Complete A Write Pattern.

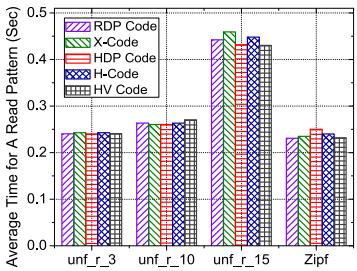
Fig. 8. Partial stripe write efficiency in reconstruct-writemode ($p = 13$).



(a) Average I/O Operations per Read Pattern in Degraded Read.



(b) Time for A Read Pattern in Degraded Read.



(c) Time for A Read Pattern in Normal Read.

Fig. 9. Read efficiency ($p = 13$).

usually needs the most number of I/O operations to complete a write pattern. For other four RAID-6 codes except HDP Code, the horizontal parity chain grants them less number of I/O operations. Moreover, we can observe that HV Code needs the least number of I/O operations for a write pattern. For example, for the trace "uniform_w_10", HV Code will reduce I/O operations by 8.7 percent (compared with RDP Code), 28.9 percent (compared with X-Code), 16.6 percent (compared with HDP Code), and 15.0 percent (compared with H-Code), respectively.

Fig. 8b shows the capability of the evaluated codes to balance the I/O distribution when conducting various write traces in reconstruct-write mode. We can find that both RDP Code and H-Code suffer from unbalanced I/O distribution. This is because both of them require dedicated disks to keep parity elements, which will not be read during the generation of new parity elements in reconstruct-write mode. On the contrary, other disks will be busy serving extra intensive read operations to calculate the new parity elements. For these five write traces, the load balancing rate of HV Code in reconstruct-write mode is 1.04, 1.02, 1.02, 1.05, and 1.03, respectively, which approach to the perfect load balancing rate (i.e., 1).

Fig. 8c presents the average time to serve a write pattern in reconstruct-write mode. We can observe that X-Code requires the longest time to complete a write pattern since it derives the maximum number of I/O operations to serve a write pattern. Meanwhile, we have an interesting finding that in reconstruct-write mode of partial stripe writes, H-Code could not preserve its optimization as in read-modify-write mode. Compared with H-Code, HV Code achieves better performance in reconstruct-write mode. For example, HV Code needs 14.3 percent less time to finish a write pattern when compared with H-Code in the write trace "uniform_w_10". Compared with X-Code, the saving will increase to 28.3 percent for the same trace.

5.2 Read Efficiency

5.2.1 Degraded Read Comparison

Evaluation method. Given a encoded file (encoded by RDP Code, X-Code, HDP Code, H-Code, and HV Code respectively), we let the elements hosted on a disk corrupted (by either injecting faults or erasing the data on that disk). We then launch five read traces including four uniform read traces "uniform_r_L" with the length of L ($L := 3, 5, 10, 15$, respectively) and one Zipf read trace. Two metrics are concerned in this failure case, i.e., the averaged time and the I/O efficiency for a degraded read pattern.

Specifically, in the running of each read pattern, suppose L' denotes the number of elements returned for a degraded read pattern. When the L requested data elements happen on the surviving disks, then $L' = L$. Otherwise, if the L requested elements include the lost elements, then the recovery of the lost elements will be triggered by fetching the associated elements and finally $L' >= L$. The needed time for a degraded read pattern is recorded from the time of issuing the degraded read pattern to the time when the L' elements are retrieved from the disk array to the main memory. We then evaluate these two metrics under the data corruption on every disk, and calculate the expectation results in Fig. 9.

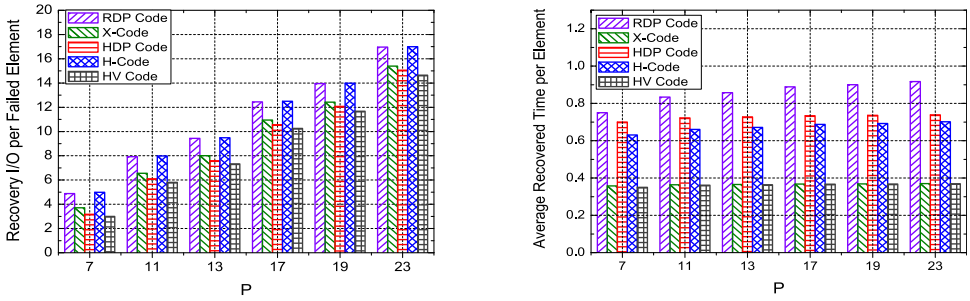
With respect to the I/O efficiency (shown in Fig. 9a), HV Code offers competitive performance by significantly reducing the number of read elements. When $L = 15$, HV Code eliminates about 16.7, 39.0, 8.3, and 12.4 percent degraded read operations compared with RDP Code, X-Code, HDP Code, and H-Code, respectively.

For the averaged time of a degraded read pattern (as shown in Fig. 9b), X-Code requires the maximum time. This observation also proves the advantage of horizontal parity when handling degraded read operations. Meanwhile, this figure also reals that it usually needs more time when the number of elements to read increases. HV Code significantly outperforms X-Code and is a bit more efficient when compared with RDP Code, HDP Code, and H-Code.

5.2.2 Normal Read Efficiency

Evaluation method. We encode a file by using RDP Code, X-Code, HDP Code, H-Code, and HV Code, respectively. After that, we first issue traces with uniform read patterns that retrieve a constant number of continuous data elements at a uniformly selected starting point. We consider three uniform read traces, i.e., "uniform_r_L", where $L := 3, 10, 15$ and L indicates the number of continuous data elements to retrieve in a read pattern. We test 100 such kind of read patterns with different lengths. Besides, we also launch Zipf trace with 200 read patterns. As the normal read will not incur extra I/O operation, we only record the average time to finish a read pattern as shown in Fig. 9c.

We have two observations from Fig. 9c. First, a read pattern with more requested data elements will usually take more time. For example, for RDP Code, the read pattern with the length of 3 will take 0.24 s on average. The needed



(a) Recovery for Single Disk Failure. (b) Recovery for Double Disk Failure.

Fig. 10. Comparison on disk failure recovery.

time will increase to 0.44 s when the number of requested data elements is 15. Second, all the five codes have similar performance on normal read. For example, when $L = 10$, HV Code takes 2.4 percent more time than H-Code to serve a read pattern. When $L = 15$, HV Code needs 4.0 percent less time compared with H-Code in reverse. This is because the normal read will not cause extra I/O operations except the requested data elements, and finally all the five codes will serve the same number of requested data elements for a trace.

5.3 The Recovery I/O for Single Disk Failure

In this test, we mainly compare the required I/O to reconstruct a failed element. An example of single disk repair in HV Code is shown in Fig. 5b when $p = 7$, in which at least 18 elements have to retrieve for the recovery of lost elements and thus it needs three elements on average to repair each lost element on the failed disk.

Evaluation method. Given an encoded file, for each case of data corruption, we let the elements on that disk corrupted, evaluate the minimal averaged elements that are retrieved from the surviving disks to recover a lost element, and calculate the expectation result.

As shown in Fig. 10a, HV Code requires the minimal number of the needed elements to repair an invalid element among the five candidates. The I/O reduction ranges from 2.7 percent (compare with HDP Code) to 13.8 percent (compare with H-Code) when $p = 23$. The saving will expand to the range from 5.4 percent (compare with HDP Code) to 39.8 percent (compare with H-Code) when $p = 7$. This superiority should own to the shorter parity chain in HV Code compared to those in other codes.

5.4 The Recovery for Double Disk Failures

Evaluation method. Suppose the average time to recover an element, either data element or parity element, is R_e and the longest length among all the recovery chains is L_c, then the time needed to finish the recovery can be evaluated by $L_c \cdot R_e$.

The comparison results are shown in Fig. 10b, which shows a big difference among the compared codes. When $p = 7$, both of X-Code and HV Code respectively reduce nearly 47.4, 47.4, and 43.2 percent of the reconstruction time when compared with RDP Code, HDP Code, and H-Code. This saving will increase to 59.7, 50.0, and 47.4 percent when $p = 23$. This huge retrenchment should owe to the placement of parity elements in X-Code and HV Code, both of which distribute two parity elements over every single disk. Every parity element will lead a recovery chain, which begins at a start point (the start point can be obtained by the chain intersecting either one of the two failed disks only) and ends at a parity element. Therefore, four recovery chains can be parallel executed without disturbing each other. Both of H-Code and HDP Code though place two parity elements over every disk, the dependent relationship between the two kinds of parities in HDP Code could extend the reconstruction time.

6 Conclusion

In this paper, we propose HV Code, which can be deployed over $p - 1$ disks (p is a prime number). HV Code evenly places parities over the disks to achieve the optimization of I/O balancing. HV Code significantly decreases the I/O operations induced by the partial stripe write in both read-modify-write mode and reconstruct-write mode. Meanwhile, the shortened length of parity chain also grants HV Code a more efficient recovery for single disk failure and good performance on degraded read operation. HV Code also accelerates the repair of two disabled disks by deriving four independent recovery chains. The performance evaluation demonstrates the efficiency brought by HV Code.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Grant Nos. 61232003, 61433008), the National High Technology Research and Development Program of China (Grant No. 2013AA013201), and the State Key Laboratory of High-end Server and Storage Technology (Grant No. 2014HSSA02). Jiwu Shu is the corresponding author of this paper. A preliminary version [1] of this paper was presented at the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14). In this journal version, we include additional analysis results and more experiment findings for HV Code.

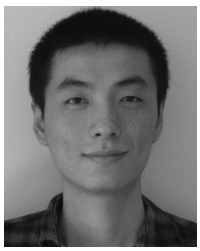
References

[1] Z. Shen and J. Shu, "HV code: An all-around MDS code to improve efficiency and reliability of RAID-6 systems," in *Proc. IEEE/IFIP 44th Annu. Int. Conf. Dependable Syst. Netw.*, 2014, pp. 550–561.

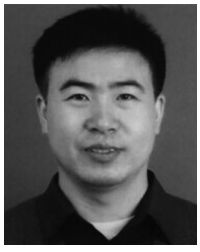
[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *ACM SIGOPS Operating Syst. Rev.*, vol. 37, pp. 29–43, 2003.

[3] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *Proc. USENIX Conf. Annu. Techn. Conf.*, 2012, p. 2.

[4] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007 p. 2.
[5] B. Schroeder and G. Gibson, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. File Storage Technol.*, 2007, pp. 1–16.
[6] C. Wu, S. Wan, X. He, Q. Cao, and C. Xie, "H-code: A hybrid MDS array code to optimize partial stripe writes in raid-6," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 782–793.
[7] R. A. DeMoss and K. B. DuLac, "Delayed initiation of read-modify-write parity operations in a raid level 5 disk array," U.S. Patent 5 388 108 A, Feb. 1995.
[8] C. Peter, "Striping in a RAID level 5 disk array," in *Proc. ACM SIGMETRICS Conf. Meas. Modeling Comput. Syst.*, 1995, pp. 136–145.
[9] A. Thomasian, "Reconstruct versus read-modify writes in raid," *Inform. Process. Lett.*, vol. 93, no. 4, pp. 163–168, 2005.
[10] W. Xiao, J. Ren, and Q. Yang, "A case for continuous data protection at block level in disk array storages," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 6, pp. 898–911, Jun. 2009.
[11] C. E. Lubbers, S. G. Elkington, and R. H. McLean, "Enhanced raid write hole protection and recovery," U.S. Patent 5 774 643 A, Jun. 1998.
[12] L. Xu and J. Bruck, "X-code: MDS array codes with optimal encoding," *IEEE Trans. Inform. Theory*, vol. 45, no. 1, pp. 272–276, Jan. 1999.
[13] 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 balancing in RAID-6," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw.*, 2011, pp. 209–220.
[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 *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, p. 1.
[15] C. Jin, H. Jiang, D. Feng, and L. Tian, "P-code: A new RAID-6 code with optimal properties," in *Proc. ACM 23rd Int. Conf. Supercomput.*, 2009, pp. 360–369.
[16] M. Blaum, J. Brady, J. Bruck, and J. Menon, "EVENODD: An efficient scheme for tolerating double disk failures in raid architectures," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 192–202, Feb. 1995.
[17] L. Xu, V. Bohossian, J. Bruck, and D. G. Wagner, "Low-density MDS codes and factors of complete graphs," *IEEE Trans. Inform. Theory*, vol. 45, no. 6, pp. 1817–1826, Sep. 1999.
[18] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, pp. 251–264.
[19] S. B. Wicker and V. K. Bhargava, *Reed-Solomon Codes and Their Applications*. Hoboken, NJ, USA: Wiley, 1999.
[20] J. S. Plank and L. Xu, "Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications," in *Proc. IEEE 5th Int. Symp. Netw. Comput. Appl.*, 2006, pp. 173–180.
[21] J. Plank, "The RAID-6 liberation codes," in *Proc. 6th USENIX Conf. File Storage Technol.*, 2008, pp. 97–110.
[22] J. S. Plank, "A new minimum density RAID-6 code with a word size of eight," in *Proc. 7th IEEE Int. Symp. Netw. Comput. Appl.*, 2008, pp. 85–92.
[23] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems," in *Proc. IEEE 6th Int. Symp. Netw. Comput. Appl.*, 2007, pp. 79–86.
[24] J. L. Hafner, "WEAVER codes: Highly fault tolerant erasure codes for storage systems," in *Proc. 4th Conf. USENIX Conf. File Storage Technol.*, 2005, p. 16.
[25] S. Wan, Q. Cao, C. Xie, B. Eckart, and X. He, "Code-M: A non-MDS erasure code scheme to support fast recovery from up to two-disk failures in storage systems," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2010, pp. 51–60.
[26] J. L. Hafner, "HoVer erasure codes for disk arrays," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2006, pp. 217–226.
[27] M. Sathiamoorthy, M. Asteris, D. Papailiopoulos, A. G. Dimakis, R. Vadali, S. Chen, and D. Borthakur, "XORing elephants: Novel erasure codes for big data," *Proc. VLDB Endowment*, vol. 6, pp. 325–336, 2013.
[28] K. M. Greenan, X. Li, and J. J. Wylie, "Flat XOR-based erasure codes in storage systems: Constructions, efficient recovery, and tradeoffs," in *Proc. IEEE 26th Symp. Mass Storage Syst. Technol.*, 2010, pp. 1–14.
[29] S. Xu, R. Li, P. Lee, Y. Zhu, L. Xiang, Y. Xu, and J. Lui, "Single disk failure recovery for x-code-based parallel storage systems," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 995–1007, Apr. 2014.
[30] M. Holland and G. A. Gibson, "Parity declustering for continuous operation in redundant disk arrays," in *Proc. ACM 5th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 1992, pp. 23–35.
[31] Z. Shen, J. Shu, and Y. Fu, "Seek-efficient I/O optimization in single failure recovery for XOR-coded storage systems," in *Proc. IEEE 34th Int. Symp. Reliable Distrib. Syst.*, Sep. 2015.
[32] L. Xiang, Y. Xu, J. Lui, and Q. Chang, "Optimal recovery of single disk failure in RDP code storage systems," in *Proc. ACM SIGMETRICS/Perform. Joint Int. Conf. Meas. Model. Comput. Syst.*, 2010, pp. 119–130.
[33] Y. Zhu, P. P. Lee, Y. Hu, L. Xiang, and Y. Xu, "On the speedup of single-disk failure recovery in XOR-coded storage systems: Theory and practice," in *Proc. IEEE 28th Symp. Mass Storage Syst. Technol.*, 2012, pp. 1–12.
[34] J. S. Plank, S. Simmerman, and C. D. Schuman, "Jerasure: A library in c/c++ facilitating erasure coding for storage applications-version 1.2," Univ. Tennessee, Knoxville, TN, USA, Tech. Rep. CS-08-627, vol. 23, 2008.
[35] 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 *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, pp. 159–172.
[36] R. Noronha, X. Ouyang, and D. Panda, "Designing a high-performance clustered NAS: A case study with pNFS over RDMA on InfiniBand," in *Proc. 15th Int. Conf. High Perform. Comput.*, 2008, pp. 465–477.
[37] RANDOM.ORG. (2010). Random integer generator [Online]. Available: http://www.random.org/integers/
[38] D. Borthakur, R. Schmidt, R. Vadali, S. Chen, and P. Kling, "HDFS raid," in *Proc. Hadoop User Group Meeting*, 2010, https://developer.yahoo.com/blogs/hadoop/hadoop-user-group-meeting-recap-november-2010-1951.html
[39] Colossus, successor to google file system [Online]. Available: http://static.googleusercontent.com/media/research.google.com/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf



Zhirong Shen received the bachelor's degree from the University of Electronic Science and Technology of China. He is currently working toward the PhD degree with the Department of Computer Science and Technology, Tsinghua University. His current research interests include storage reliability and storage security.



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

Yingxun Fu received the bachelor's degree from North China Electric Power University in 2007, the master's degree from Beijing University of Posts and Telecommunications in 2010, and the doctor's degree from Tsinghua University in 2015. His current research interests include storage reliability and distributed systems.