6-2009

P-Code: A New RAID-6 Code with Optimal Properties

Chao Jin
Huazhong University of Science & Technology, chjinhust@gmail.com

Hong Jiang
University of Nebraska - Lincoln, jiang@cse.unl.edu

Dan Feng
Huazhong University of Science & Technology, dfeng@hust.edu.cn

Lei Tian
University of Nebraska - Lincoln, ltian2@unl.edu

Published in *ICS '09: Proceedings of the 23rd International Conference on Supercomputing*, June 8-12, 2009, Yorktown Heights, New York, USA, pp. 360–369.

P-Code: A New RAID-6 Code with Optimal Properties

Chao Jin,[1] Hong Jiang,[2] Dan Feng,[1] Lei Tian,[1, 2]

1. Key Laboratory of Data Storage Systems, Ministry of Education of China,
School of Computer Science & Technology, Huazhong University of Science & Technology

2. Department of Computer Science & Engineering, University of Nebraska-Lincoln

Corresponding author — D. Feng

E-mail addresses — dfeng@hust.edu.cn / chjinhust@gmail.com / jiang@cse.unl.edu / tian@cse.unl.edu / ltian@hust.edu.cn

Abstract

RAID-6 significantly outperforms the other RAID levels in disk-failure tolerance due to its ability to tolerate arbitrary two concurrent disk failures in a disk array. The underlying parity array codes have a significant impact on RAID-6's performance. In this paper, we propose a new XOR-based RAID-6 code, called the Partition Code (P-Code). P-Code is a very simple and flexible vertical code, making it easy to understand and implement. It works on a group of (prime – 1) or (prime) disks, and its coding scheme is based on an equal partition of a specified two-integer-tuple set. P-Code has the following properties: (1) it is a Maximum-Distance-Separable (MDS) code, with optimal storage efficiency; (2) it has optimal construction and reconstruction computational complexity; (3) it has optimal update complexity (i.e., the number of parity blocks affected by a single data-block update is minimal). These optimal properties of P-Code are proven mathematically in this paper. While X-Code is provably optimal and RDP is proven optimal in computational complexity and storage efficiency, the latter in its current form is not optimal in up-date complexity. We propose a row-parity placement strategy for RDP to help it attain optimal update complexity. P-Code complements the other two optimal RAID-6 codes, X-code and the tweaked RDP, to provide a near-full set of optimal RAID-6 con-figurations of typical disk-array size (e.g., 4-20 disks). That is, for any prime in a typical array size range, P-code can be deployed for (prime – 1) disks optimally, while X-code (or P-Code) and the tweaked RDP can be respectively deployed for (prime) and (prime + 1) disks optimally. Moreover, P-code's potentially beneficial properties such as the flexible association between the blocks and their labels may find useful applications in distributed environments.

Keywords: operating systems, reliability, performance, algorithms

1. Introduction

RAID (Redundant Array of Independent Disks) techniques [1] are widely used in modern storage systems to achieve high performance and reliability. By maintaining redundant information within an array of hard disks, RAID can protect data against one or more disk failures. Among the commonly used RAID levels, RAID-1, RAID-4, and RAID-5 can tolerate exactly one single-disk failure; another disk failure will lead to permanent data loss. RAID-10 and RAID-01 tolerate multiple failures by mirroring disks into pairs, but two concurrent disk failures in any mirroring pair will also lead to permanent data loss; moreover, their storage efficiency, at only 50%, is quite low. RAID-6 is specified to tolerate any two concurrent disk failures in a disk array, thus providing a higher level of data reliability.

While RAID-5 and RAID-10 have been most commonly deployed in production data centers, RAID-6 has received increasing attention from the academia and industry recently and is poised to be more widely deployed in data centers to increase data reliability and integrity. The reason behind this increasing interest in the RAID-6 architecture is twofold. On the one hand, recent findings from real world by researchers [2] have reported that partial or complete disk failure rates are actually much higher than previously and commonly estimated, which suggests an urgent need to significantly improve the reliability of RAID systems. On the other hand, while the number and capacity of disks have been growing almost exponentially [3], individual disk failure rates remain largely unchanged, which means that in supercomputing data centers, where there are thousands of hard disks and two or more concurrent disk failures are no longer rare, the ability to tolerate double disk failures becomes ever more important.

RAID-6's double-disk-failure recovery ability is implemented through the underlying erasure codes, such as Reed-Solomon [4] and EVENODD [5]. The performance of a RAID-6 array is largely determined by the erasure code used. In general, we measure a RAID-6 code's performance in terms of its computational complexity, update complexity and storage efficiency. Computational complexity is proportional to the CPU computational overhead during construction and reconstruction. Update complexity indicates the average number of parity blocks affected by an update (write) of a single data block [6, 9]. For RAID-6, every data block is protected by at least two distinct parity blocks on average, so the optimal update complexity is 2. The update complexity can significantly influence the write performance of RAID-6, espe-

cially for small writes. On the other hand, storage efficiency is also an important metric, which measures the percentage of storage space used by the parity blocks to protect the data blocks. The smaller the percentage is, the higher the storage efficiency is. Generally, for any erasure code, its error-correcting ability and redundant rate satisfy the Singleton formula [7]. Particularly, if it attains the Singleton bound, we call it a Maximum-Distance-Separable (MDS) code. In other words, it attains the optimal storage efficiency.

In this paper, we propose a new and efficient RAID-6 architecture by defining a new erasure code, called the Partition Code, or simply P-Code. P-Code is a vertical code, in which its parities are calculated only with XOR operations and spread evenly across component disks. Unlike horizontal codes that concentrate parity blocks on extra dedicated parity disks, such as EVENODD [5], this feature of P-Code results in balanced disk accesses during the write-dominated periods. We will further discuss the implementation issues in Section 6.4. P-Code has the optimal update complexity of 2, for each data block contributes to the calculations of, and is protected by, exactly 2 parity blocks. We will prove in Section 4 that P-Code is an MDS code with a *minimal column distance* of 3, and thus has the optimal storage efficiency among all the codes that are capable of protecting against double disk failures. We will derive in Section 6 the lower bound of construction and reconstruction computational complexity for any form of MDS RAID-6 codes, and prove that P-Code has attained this bound. We use a very simple label-oriented approach to describe the construction algorithm, the proof of MDS property, and the reconstruction algorithm of P-Code, so it can be easily understood and implemented in RAID-structured storage systems.

This paper makes the following important contributions:

1. We define the optimality of RAID-6 codes in terms of *optimal computational complexity*, *optimal update complexity*, and *optimal storage efficiency*. We describe and prove these optimal properties mathematically.
2. We analyze two state-of-the-art RAID-6 codes, X-Code and RDP, and prove that X-Code is an optimal RAID-6 code. RDP in its current form is optimal in computational complexity and storage efficiency, but non-optimal in update complexity [8, 13]. We propose a row-parity placement strategy for RDP to help it attain optimal update complexity (See section 6.2).
3. We propose a new optimal RAID-6 code, called the Partition Code (P-Code). P-Code complements the other two optimal RAID-6 codes: X-code and the tweaked RDP. That is, for any prime in a typical disk-array size range, P-code can be deployed for (prime – 1) disks optimally, while X-code (or P-Code) and the tweaked RDP can be respectively deployed for (prime) and (prime + 1) disks optimally.
4. We further explore the potential beneficial properties of P-Code. For instance, its flexible association between the labels and the data units may find potential applications in distributed storage environments.

The rest of this paper is organized as follows. The next section discusses the related work. We present a detailed description of the label-oriented construction algorithm, proof of MDS property, and reconstruction algorithm for P-Code with (prime–1) disks in Section 3, Section 4, and Section 5 respectively. Then we analyze the properties and performance of P-Code, X-Code, and RDP in Section 6. In Section 7 we present the construction of P-Code with (prime) disks and in Section 8 we extend P-Code to arbitrary disk array size. We conclude this paper in Section 9 with thoughts on future works.

2. Related Work

Erasure coding techniques [11] are widely used in many fields such as telecommunication and data storage for their error-correcting capability. There are several well-known erasure coding schemes that are capable of protecting against two or more disk failures in an array of disks, such as Reed-Solomon coding and parity array coding. Reed-Solomon coding [12] is a powerful erasure coding technique with strong error-correcting capability. It uses Galois Field arithmetic during encoding and decoding. Galois Field addition is equivalent to XOR, but its multiplication is much more complicated, usually involves a table-lookup operation that alleviates the computation intensity. Unlike Reed-Solomon coding, parity array coding depends solely on XOR operations during encoding and decoding. This simplicity makes them more preferable for RAID storage systems. Most parity array codes, such as EVENODD, RDP, and X-Code, can tolerate two disk failures. STAR [18] and the codes presented in [19] and [20] are MDS parity array codes that can tolerate more than two disk failures, and they can be regarded as the extensions of EVENODD in the higher dimension. There are some other codes that can also tolerate multiple disk failures, such as R5X0 [6], HoVer [21], and WEAVER [22], but they are non-MDS codes. Since P-Code is an MDS parity array code that can tolerate double disk failures, we only review codes in the same category in the following, where p is a prime number, as follows.

EVENODD. EVENODD code for a $(p + 2)$-disk array is defined in a $(p - 1)$-row-by-$(p + 2)$-column matrix. The first disks store data blocks and the last two disks store parity blocks from parity chains across the data disks along slope 0 and slope 1 respectively. Note that there are p diagonals along slope 1, so one parity block is XORed into the other $p - 1$ blocks as an adjusting factor S, and is not actually stored. EVENODD is not optimal in either computational complexity or update complexity.

RDP. RDP code for a $(p + 1)$-disk array is defined in a $(p - 1)$-row-by-$(p + 1)$-column matrix. Its first p disks store data and row-parity blocks, and the last disk stores diagonal-parity blocks from parity chains across the first p disks. RDP makes an improvement upon EVENODD because it successfully avoids calculating the adjusting factor S, and attains the optimal computational complexity. But RDP in its current form fails to attain the optimal update complexity [8, 13]. In section 6.2, we will present a row-parity block placement strategy for RDP to help it achieve optimal update complexity.

X-Code. X-Code for a p-disk array is defined in a $p \times p$ matrix. Data blocks are stored in the first $p - 2$ rows. The last two rows store the parity blocks, calculated from the parity chains along slope 1 and slope -1 respectively. The coding scheme shapes like the letter "X" in geometry, hence the name. X-Code is a vertical code with the optimal computational complexity and update complexity.

Liberation Codes. Plank proposed a horizontal code called the Liberation Codes in [13] for a $(p + 2)$-disk array. The coding process can be described by the multiplication of a specified Binary Distribution Matrix (BDM) and a data vector. The BDM of Liberation Codes has the minimal number of ones, indicating that Liberation Codes has optimal update complexity among all RAID-6 horizontal codes. However, its computational complexity does not reach optimal, though the penalty over optimal is small, due to the fact that each data block participates in more than two parity chains on average.

Our P-Code in this paper has similar structure as the lowest density codes originally proposed by Zaitsev et al. in [14], but with significant differences in the construction of the parity chains. While the forerunners used either graph theory [15] or generator matrix (parity-check matrix) [7] to describe the construction process of the lowest density codes, which are probably hard for the RAID designers to follow, we use a complete label-oriented approach to describe the construction algorithm, the proof of MDS property, and the reconstruction algorithm of P-Code, which can be easily understood and straightforwardly implemented. We will also show that P-Code has many variations because of its flexible association between the labels and the data units (See section 6.3). Cassuto et al. proposed the cyclic lowest density codes for (prime – 1) disks [16], in which a codeword (i.e., an instance of the code) can be obtained by cyclically shifting the columns of another codeword. In order to maintain this property, their generator matrices are restricted. Nanda et al. also proposed a RAID-6 algorithm for (prime – 1) disks in [17], but we will show that P-Code can be deployed optimally for not only (prime – 1) disks but also (prime) disks (See section 7). Additionally, the optimal properties of the lowest density codes were not fully explored or clearly stated in the past literature. We will define the optimality of RAID-6 code in details and prove that P-Code is an optimal RAID-6 code with optimal computational complexity, optimal update complexity and optimal storage efficiency. We will discuss these features respectively in the following sections.

3. Construction Of P-Code With Prime – 1 Disk

P-Code for a $(p - 1)$-disk array is defined in a $(p - 1)/2$-row-by-$(p - 1)$-column matrix, where p is a prime number greater than 3. We will show later how the primality restriction guarantees the MDS property of P-Code.

3.1. Data/Parity Block Labeling

First we label the $p - 1$ disks sequentially by $d_1, d_2, \ldots, d_{p-1}$. For each disk, the first block is assigned to be a parity block and the remaining $(p - 3)/2$ blocks are assigned to be data blocks. Each parity block is labeled with an integer equal to the index of the disk on which it resides, that is, the parity block on the first disk is labeled with (1) and on the last disk with $(p - 1)$. Each data block is labeled with an unordered two-integer-tuple (m, n) for (m, n) in the set C defined in Equation (1) below.

$$C = \{(m, n) \mid (1 \le m, n \le p - 1), m \ne n, m + n \ne p\} \quad (1)$$

d_1	d_2	d_3	d_4	d_5	d_6
(1)	(2)	(3)	(4)	(5)	(6)
(2,6)	(3,6)	(1,2)	(1,3)	(1,4)	(1,5)
(3,5)	(4,5)	(4,6)	(5,6)	(2,3)	(2,4)

Figure 1. Block labeling for P-Code with (prime – 1) disks ($p = 7$). The parity block in each disk is labeled with a single integer that equals the index of the disk. The data blocks in each disk are each labeled with a two-integer-tuple, such that the modular p addition of the two integers in each tuple equals the index of the disk.

```
STEP 1. Label all the data and parity blocks in the structure
        of P-Code.
STEP 2. do {
        select an unconstructed parity block labeled with
          integer (i);
        select from all the disks the data blocks whose
          labels include integer i;
        construct the selected parity block by XORing
          all the selected data blocks together;
        } until (all the parity blocks are constructed)
```

Figure 2. Construction Algorithm of P-Code

It is easy to see that there are $(p - 1)(p - 3)/2$ elements in C. We define $p - 1$ subsets of C denoted by $C_1, C_2, \ldots C_{p-1}$, where C_i $(1 \le i \le p - 1)$ is defined as in Equation (2).

$$C_i = \{(m, n) \mid (m, n) \in C, m + n \equiv i \bmod p\} \quad (2)$$

After some steps of simple mathematical reasoning, we can come to the following conclusions.

1. Any element in set C can be found in some subset C_i. That is,

$$C = \bigcup_{i=1}^{p-1} C_i \quad (3)$$

2. For any two different subsets C_i and C_j, they do not share any common element, namely,

$$C_i \cap C_j = \varnothing \ (i \ne j) \quad (4)$$

3. All the subsets have an equal number of $(p - 3)/2$ elements.

Clearly, $\{C_1, C_2, \ldots C_{p-1}\}$ is an *equal partition* of set C. P-Code is defined on this *partition*.

Now let's return to the labeling of data blocks. We label each data block in disk d_i with an element in subset C_i, $1 \le i \le p - 1$. Since there are exactly $(p - 3)/2$ data blocks in and d_i and $(p - 3)/2$ elements in C_i, each element in C_i can be assigned to a different data block in d_i, and vice versa. Figure 1 shows the labeling for a 6-disk array ($p = 7$).

3.2. Construction Process

The Construction procedure is designed to construct all the parity blocks from data blocks. With all the parity and data blocks labeled as above, the description of the P-Code construction procedure becomes straightforward. We present the construction algorithm for P-Code in Figure 2.

In the P-Code structure, parity and data blocks with a common integer in their labels form an independent parity chain, which we represent as $P(i)$. For instance, in Figure 1, parity block (1) on disk d_1 is constructed by data blocks (1, 2) on d_3, (1, 3) on d_4, (1, 4) on d_5, and (1, 5) on d_6, together they form the parity chain $P(1)$. Obviously, there are $p - 1$ parity chains in a codeword of P-Code.

From the structure of P-Code we can find an interesting phenomenon, that is, in each disk, the labels contain each integer between 1 and $p - 1$ exactly once except one integer, which we call the *missing number* of that disk. For example, 4 is the *missing number* of disk d_1 as shown in Figure 1. For each integer u between 1 and $p - 1$, we can derive from Equation (2) that the corresponding integer v which forms a label (u, v) in d_i with u satisfies $v = i - u \bmod p$. But there are two exceptions. First, the corresponding integer for $u = i$ is $v = 0$, but 0 is not in the domain of the labels, so this singleton (i) is not in C_i, and is assigned to be the label of the parity block in disk d_i. Second, there is one case that the corresponding integer of u is u itself, but tuple (u, u) is also not in C_i. In fact, this integer u in the second exception is exactly the *missing number* of disk d_i. Let m_i denote the *missing number* of disk d_i, we have the following equation.

$$2m_i \equiv i \bmod p \quad \text{or} \quad m_i = \begin{cases} i/2 & (\text{if } i \text{ is even}) \\ (i+p)/2 & (\text{otherwise}) \end{cases} \tag{5}$$

Excluding the above two exceptions, there are $(p - 3)$-tuples that satisfy Equation (2). Since the tuples are unordered, there are actually $(p - 3)/2$ elements in subset C_i, which is consistent with our previous conclusion. P-Code does not restrict the corresponding relationships between the elements in C_i and the data blocks in disk d_i. In other words, data blocks within the same disk can exchange their labels without sacrificing the fault-tolerant capability of the array. In a practical implementation, the labels can be arranged to the data blocks in a particular order (e.g., ascending) so that we can quickly locate a data block in a disk when given its label. This property is further discussed in Section 6.3.

4. Proof of P-Code's MDS Property

We will prove P-Code is a Maximum-Distance-Separable code in this section. In order for the proof to be more specific, we use the terms of *columns* and *symbols* to represent the *disks* and *blocks* respectively.

P-Code is a linear code. Let d denote its *minimal column distance*. According to the structure of P-Code, we can derive from the Singleton formula [7] that $d \le 3$. In order to prove P-Code's MDS property, we must prove that it attains the Singleton bound, i.e., $d = 3$. For a linear code, its *minimal column distance* is equal to its *minimal column weight*, so the proof can be done by proving that P-Code has a *minimal column weight* of 3, i.e., a valid codeword of P-Code has at least three non-zero columns (a non-zero column means that at least one symbol in that column is not zero). We start this proof with the following Lemma.

Lemma 1. *In the sequence of integers* $\{a(k) \mid k = 0, 1, ..., p - 1\}$,

$$a(k) = \left\{(-1)^k \left(k + \tfrac{1}{2}\right)(n_1 - n_2) + \frac{(n_1 + n_2)}{2}\right\} \bmod p$$

where p is prime and $0 \le n_2 < n_1 \le p - 1$, each integer between 0 and $p - 1$ occurs exactly once, with n_1 and n_2 to be the two endpoints.

Proof: First we observe that the p integers in the sequence are all between 0 and $p - 1$ since the operation is modulus p arithmetic. Thus, if any two integers are not equal to each other, each of the integers 0, 1, …, $p - 1$ must appear exactly once in the sequence, in other words, the sequence is a permutation of the integers 0, 1, …, $p - 1$.

Let's select from the sequence two arbitrary integers, denoted as $a(k_1)$ and $a(k_2)$ respectively. Then we get:

$$a(k_1) - a(k_2) = \left\{((-1)^{k_1}\left(k_1 + \tfrac{1}{2}\right) - (-1)^{k_2}\left(k_2 + \tfrac{1}{2}\right))(n_1 - n_2)\right\} \bmod p$$

The second part, $(n_1 - n_2)$, is between 1 and $p - 1$, and is not divisible by p. Since p is prime, we can prove by an exhaustive search (i.e., assume k_1 or k_2 is even or odd) that the first part, $((-1)^{k_1}(k_1 + \tfrac{1}{2}) - (-1)^{k_2}(k_2 + \tfrac{1}{2}))$, is also not divisible by p. Thus, $a(k_1) \ne a(k_2)$.

Now we prove that P-Code has the MDS property.

Theorem 1. *P-Code has a minimal column weight of 3, i.e., it is MDS, given that p is prime.*

Proof: Let w denote P-Code's *minimal column weight*. From the structure of P-Code, it is obvious that each parity chain overlaps with any column at most once, so it is impossible for a valid code-word of P-Code to have only one non-zero column, for otherwise some parity chains may be in a parity-inconsistent state. Thus, $w > 1$.

Suppose $w = 2$, i.e., a codeword of P-Code may have only two non-zero columns. We assume that column d_i and d_j $(1 \le i, j \le p - 1)$ are the two non-zero columns, and all the rest columns have only zero symbols. In the subsequent statements, $P(i, j)$ denotes the symbol of parity chain $P(i)$ located in disk d_j.

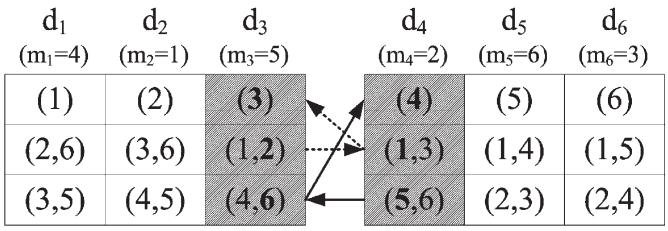

Figure 3. Illustration of the *recovery chain* for disk d_3 and d_4 in the structure of a 6-disk P-Code. The corresponding *recovery chain* for the two disks is 5-6-4-0-3-1-2. The solid line with arrows depicts the traversing starting from parity chain P(5) and block (5,6), then P(6) and (4,6), and then P(4) and (4). The dotted line with arrows depicts the traversing starting from parity chain P(2) and block (1,2), then P(1) and (1,3), and then P(3) and (3). The integers in bold in each block label shows the index of the parity chain used to recover that block. The *recovery chain* illustrates the order of the parity chains used to traverse all the blocks in the two failed disks.

STEP 1. Compute the *missing number* m_i of disk d_i, and m_j of d_j, using Equation (5).
STEP 2. Compute the *recovery chain* of disk d_i and d_j, with m_i to be the start point and m_j to be the endpoint.
STEP 3. Start two reconstruction threads, running in parallel.
Thread 1: Fetch the start point m_i of the *recovery chain*, $u \leftarrow m_i$.
while ($u \neq 0$) {
reconstruct the lost block in parity chain $P(u)$ using the other blocks in the same parity chain.
fetch the next integer u along the *recovery chain*.
}
Thread 2: Fetch the end point m_j of the recovery chain, $v \leftarrow m_j$.
while ($v \neq 0$) {
reconstruct the lost block in parity chain $P(v)$ using the other blocks in the same parity chain.
fetch the next integer v reversely along the *recovery chain*.
}

Figure 4. Reconstruction Algorithm of P-Code for Double-Disk Failures.

First, we observe that symbol $P(m_i, i)$ do not exist, since m_i is the *missing number* of column d_i. Then, we can see that symbol $P(m_i, j)$ must be zero, since all the other symbols of $P(m_i)$ are located in the zero columns. According to Equation (2), this symbol can also be denoted as $P(|(j - m_i) \bmod p, j)|$. Next, we can see that symbol $P(|(j - m_i) \bmod p|, i)$ must also be zero. Similarly, we can see that symbol $P((i - j + m_i) \bmod p, j)$ is also zero. In fact, from the repetition of this saw-like derivation we can come to the conclusion that all symbols in column d_i and d_j should be zero.

Now let's check into the process of the derivation. The first parity chain we use is $P(m_i)$, then $P((j - m_i) \bmod p)$, and then $P((i - j + m_i) \bmod p)$, and so on. The indices of the parity chains form a sequence of integers, and we call this sequence the *recovery chain*. According to Equation (5), the *recovery chain* can be transformed into m_i, $(2m_j - m_i) \bmod p$, $(3m_i - 2m_j) \bmod p$, …, m_j. Given m_i as n_1 and m_j as n_2, the *recovery chain* is exactly the same as the sequence in Lemma 1. Thus, according to Lemma 1, the *recovery chain* is a permutation of the integers from 0 to p –1, with m_i and m_j to be the two endpoints, and 0 is somewhere between m_i and m_j .

Lemma 1 guarantees that every symbol in columns d_i and d_j can be traversed exactly once along the *recovery chain*. It must be noted that there is a 0 in the *recovery chain*, that's a breakpoint. In fact, the integer 0 breaks the *recovery chain* into two parts. From the start point m_i, we go along the *recovery chain* to traverse the parity chains and the symbols in them. We stop when we reach the breakpoint of 0. Then, from the end point m_j, we go reversely along the *recovery chain* to traverse the rest parity chains and symbols. In this way, we can traverse all the parity chains and all the symbols in columns d_i and d_j.

Figure 3 illustrates the *recovery chain* of columns d_3 and d_4 in a 6-column P-Code. The corresponding *recovery chain* is 5−6−4−0−3−1−2.

As we can see, if a codeword of P-Code has only two non-zero columns, these two columns must also be zero. This is a contradiction. As a result, P-Code's *minimal column weight w* ≠ 2. Thus $w \geq 3$, but it is easy to find a codeword of column weight 3, so $w = 3$.

5. Reconstruction of P-Code

In this section, we present the P-Code reconstruction algorithms for two situations: double-disk failures and single-disk block corruptions. When disk failures occur, the disk driver can easily detect which disk fails and reports to the RAID controller that all the data/parity blocks on the failed disk become inaccessible. On the other hand, when disk block corruptions occur, the system has no idea about which disk is involved and which blocks have corrupted.

5.1. Reconstruction Algorithm for Double-Disk Failures

Suppose two of the $(p - 1)$ disks have failed in a P-Code driven disk array. Since we are aware of the indices of the two disks, we can easily construct the *recovery chain* of them. The *recovery chain* illustrates the order of the parity chains used to traverse all the blocks in the failed disks; namely, we respectively go along and reversely along the *recovery chain* from the start point and end point of it, until we reach the break point of 0. Thus, we can reconstruct all the blocks on the failed disks step by step. This feature has been shown in Section 4, which

is actually a simple illustrative example of P-Code reconstruction algorithm for double disk failures. We present the reconstruction algorithm in details as shown in Figure 4. (The failed disks are denoted as d_i and d_j.)

In Step 2 of the algorithm, given m_i and m_j, we can construct the elements of the *recovery chain* one by one using Equation (2). This method is not optimal and we have a shortcut as follows. Note that Lemma 1 has given the general expression of the elements in the *recovery chain*. Using this expression, we can quickly construct the *recovery chain* for any two failed disks. It must be noted that the correctness of this algorithm can be deduced from the proof of P-Code's MDS property.

5.2. Reconstruction Algorithm for Single-Disk Block Corruptions

To recover from single-disk block corruptions, the first and also the key step is to find which disk has corrupted blocks. Once the location of the disk is found, we can further figure out which blocks have corrupted on that disk. Then the corrupted blocks can be easily re-calculated by the corresponding parity chains.

For a P-Code driven disk array, we first compute the *verification variables* for each of the (p−1) parity chains, with the *verification variable* defined as follows.

$$V[i] = \text{XOR result of all the blocks in parity chain } P(i). \quad (6)$$

Obviously, if there is no block corrupted in parity chain $P(i)$, its *verification variable* $V[i]$ must be zero, otherwise, it must be non-zero. By checking the *verification variables*, we can figure out all the corrupted parity chains that have corrupted blocks in them. With the indices of the corrupted parity chains, we can easily find out the index of the corrupted disk. This is because the indices of the corrupted parity chains exactly make up of the labels of the corrupted blocks on the disk, and the labels of the blocks have internal relationships with the index of the disk (See Figure 1). When the index of the corrupted disk is figured out, we can get the labels of the corrupted blocks without striking a blowing. Figure 5 gives an illustrative example of this process. The reconstruction algorithm for single-disk block corruptions is presented in details in Figure 6.

d_1	d_2	d_3	d_4	d_5	d_6
(1)	(2)	(3)	(4)	(5)	(6)
(2,6)	(3,6)	(1,2)	(1,3)	(1,4)	(1,5)
(3,5)	(4,5)	(4,6)	(5,6)	(2,3)	(2,4)
V[1]=0	V[2]=0	V[3] ≠ 0	V[4] ≠ 0	V[5]=0	V[6] ≠ 0

Figure 5. Suppose the two blocks under shadow are corrupted. By checking the *verification variable* of each parity chain, we find that parity chain P(3), P(4), and P(6) have corrupted. The indices of the corrupted parity chains, 3, 4, and 6, must form the labels of the corrupted blocks in the corrupted disk, namely, a corrupted data block and a corrupted parity block. Next, suppose (i, j) is the label of the data block and (k) is the label of the parity block. From the construction process of P-Code we know that i + j ≡ k mod 7, that is, i + j + k ≡ 2k mod 7 or 13 ≡ 2k mod 7. Thus, the index of the corrupted disk is k = 3, and the corrupted blocks are (3) and (4,6).

STEP 1. Compute the *verification variable* of each parity chain.
STEP 2. Find the indices of the corrupted parity chains by checking the *verification variables* of the parity chains.
STEP 3. Find the index of the corrupted disk through the indices of the corrupted parity chains.
STEP 4. Reconstruct each corrupted block using the other blocks in the same parity chain.

Figure 6. Reconstruction Algorithm of P-Code for Single-Disk Block Corruptions.

6. Performance and Properties Analysis

In this section we will analyze the performance of P-Code and the other two relevant RAID-6 codes, X-Code and RDP. We will prove that all the three codes have attained optimal construction/reconstruction computational complexity. We will also show that P-Code and X-Code, but not RDP in its current form, have optimal update complexity. We present a row-parity block placement strategy for RDP to help it attain optimal update complexity. Then, we will explain the flexibility of P-Code in details. We assume that p is a prime number greater than 3 in the following statements.

6.1. Construction/Reconstruction Computational Complexity

From the structure of P-Code, we can see that there are (p − 3) data blocks and one parity block in each of the (p – 1) parity chains. Thus, each parity block is constructed from the data blocks in the same parity chain by (p – 4) XOR operations, and a total number of (p − 1)(p − 4) XOR operations are needed to construct all the parity blocks. Since there are (p – 1)(p – 3)/2 data blocks in all, so the construction computational complexity is 2 - 2/(p − 3) XOR operations per data block. On the other hand, when double disk failures occur, each lost data or parity block is reconstructed from the other blocks in the same parity chain by (p − 4) XOR operations, so the reconstruction computational complexity is (p − 4) XOR operations per lost block.

Similarly, we can derive out that the construction computational complexity for X-Code and RDP are 2 – 2/(p − 2) and 2 – 2/(p − 1) XOR operations per data block respectively, and the reconstruction computational complexity for X-Code and RDP are (p − 3) and (p − 2) XOR operations per lost block respectively.

Theorem 2. *P-Code, X-Code, and RDP all have attained the optimal construction/reconstruction computational complexity among any form of MDS RAID-6 codes (i.e., horizontal, vertical, or hybrid codes).*

Proof: Consider an arbitrary MDS RAID-6 Code C that is defined in a m-row-by-n-column matrix. There are $m \times n$ blocks in the structure of C. Suppose x of them are data blocks, and the remaining $m \times n - x$ are parity blocks. Since C is a RAID-6 code that can tolerate two-disk failures, its *minimal column distance* $d = 3$. Additionally, it is a MDS code, so it has attained the Singleton bound. From the Singleton formula [7], we can ascertain that C satisfies the following equation.

$$\frac{x}{m} = n - 2 \tag{7}$$

For the optimal case of the RAID-6 codes, each data block participates in the construction of exactly two parity blocks, and the parity blocks are independent from each other. In this situation, the $m \times n - x$ parity blocks are constructed from $2x$ data blocks (since each of the x data blocks participates in the construction twice). Thus, a total number of $2x - (m \times n - x) = 3x - m \times n$ XOR operations are required in the construction process. That is to say, the optimal construction computational complexity, in terms of average number of XOR operations per data block, is as follows.

$$\frac{3x - m \times n}{x} \tag{8}$$

On the other hand, when double disk failures occur, the disk array should reconstruct the $2m$ lost blocks in the two failed disks. In the structure of Code C under the optimal situation, there are $m \times n - x$ parity chains, each of which contains one parity block. According to Equation (7), $m \times n - x = 2m$. That is to say, in the reconstruction process, the $m \times n - x$ lost blocks are reconstructed from the $m \times n - x$ parity chains, with each parity chain used exactly once to reconstruct one lost block in it. Similar as the construction process, a total number of $3x - m \times n$ XOR operations are required in the reconstruction process. So the optimal reconstruction computational complexity, in terms of average number of XOR operations per lost block, is as follows.

$$\frac{3x - m \times n}{m \times n - x} \tag{9}$$

From the above three equations, we can easily come to the conclusion that P-Code, X-Code, and RDP all have attained the optimal construction/reconstruction computational complexity. The only difference is that their optimality each corresponds to a different array size , namely, $(p - 1)$, p, and $(p + 1)$ for P-Code, X-Code, and RDP respectively. ◘◘◘

6.2. Row-Parity Placement Strategy for RDP to Attain Optimal Update Complexity

From the structure of P-Code and X-Code, we can see that there is exactly one parity block in each parity chain. The parity blocks in different parity chains are independent from each other, and update of one will not affect another. Since each data block participates in exactly two distinct parity chains, one data block update triggers exactly two parity block updates. This is optimal for parity array codes that protect against two disk failures.

The update complexity of RDP is determined by the placement of the row-parity blocks. For a $(p + 1)$-disk array using RDP, the first p disks store data and row-parity blocks, and the last disk stores diagonal-parity blocks from parity chains across the first p disks. The RDP array will not attain the optimal update complexity of 2 if we simply assign the first p disks in a RAID-4 or RAID-5 style, for some row-parity blocks also participate in diagonal-parity chains. But note that there is a *missing diagonal-parity chain* in RDP's structure, that is, in each row there is one block that does not participate in any diagonal-parity chain. If we set this block to be the row-parity block in each row, the row and diagonal parity blocks will be separated from each other and the update complexity of RDP will reach optimal. In this form, for the first p disks (i.e., not including the last diagonal-parity disk) in an RDP array, the first disk is an all-data disk, and the row-parity blocks are distributed evenly across the remaining $p - 1$ disks.

6.3. Flexibility of P-Code

We have seen that the labeling of P-Code is based on an equal partition of a specified two-integer-tuple set, with each subset corresponding to a disk in the P-Code structure. Pay attention to the fact that the label of a block is not based on the coordinate of the block in the P-Code structure. That is, the blocks and their labels have no inherent relationships. The only restriction is that the blocks in the same disk should be labeled with the elements from the same subset defined in Equation (2).

Furthermore, the relationship between the disks and the subsets is also not restricted. That is, we can assign the subsets to the disks optionally without harming P-Code's correctness.

The above unique properties of P-Code lead to the flexible association between the blocks and the labels within the P-Code structure. For instance, we can move the blocks with labels within each disk, or we can even move the whole disks within the P-Code structure. All of the resulting structures can be regarded as the variations of P-Code and the properties of P-Code still hold true for them.

6.4. Other Implementation Issues

The distinct difference between horizontal codes and vertical codes lies in the placement of the parity blocks. For horizontal RAID-6 codes, the parity blocks are held on two dedicated parity disks; while for vertical RAID-6 codes, the parity blocks are spread across the data disks. This feature of the vertical codes results in balanced accesses among the disks at the write-dominated periods. Also, only the vertical codes have the possibility to attain the optimal (i.e., lowest) update complexity of 2; the liberation codes have attained the optimal update complexity among all the horizontal RAID-6 codes, but its update complexity is still larger than 2. However, the horizontal codes also have their own benefits. They have the nice feature of altering the number of data disks. On the one hand, you can shorten the array size easily. You can simply assume some data disks contain only imaginary zero and take them away from the disk array. This will not affect the remaining part of the array, since there are no parity blocks on the data

d_1 (m_1=4)	d_2 (m_2=1)	d_3 (m_3=5)	d_4 (m_4=2)	d_5 (m_5=6)	d_6 (m_6=3)	d_7 (m_7=0)
(1)	(2)	(3)	(4)	(5)	(6)	(1,6)
(2,6)	(3,6)	(1,2)	(1,3)	(1,4)	(1,5)	(2,5)
(3,5)	(4,5)	(4,6)	(5,6)	(2,3)	(2,4)	(3,4)

Figure 7. Block labeling for P-Code with (prime) disks (p = 7). The last disk d_7 is assigned to be an all-data disk. Each block in this disk is labeled with a two-integer-tuple, with the modular p addition of the two integers in each tuple to be 0. If we assign the *missing number* of the last disk to be 0, we can still construct the *recovery chain* for the last disk and any other disk in the array according to Lemma 1. The solid line with arrows depicts the *recovery chain* for the last disk and disk d_6.

d_1 (m_1=6)	d_2 (m_2=1)	d_3 (m_3=7)	d_4 (m_4=2)	d_5 (m_5=8)	d_6 (m_6=3)	d_7 (m_7=9)	d_8 (m_1=4)	d_9 (m_2=10)	d_{10} (m_3=5)
(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
(2,10)	(3,10)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)	(1,9)
(3,9)	(4,9)	(4,10)	(5,10)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(4,8)	(5,8)	(5,9)	(6,9)	(6,10)	(7,10)	(3,4)	(3,5)	(3,6)	(3,7)
(5,7)	(6,7)	(6,8)	(7,8)	(7,9)	(8,9)	(8,10)	(9,10)	(4,5)	(4,6)

Zero Block Reserved Block

Figure 8. Structure of non-standard P-Code with 8 disks. We first construct a standard P-Code with 10 disks, then set the last two disks d_9 and d_{10} to be all-zero disks, and then set all the blocks in parity chain P(9) and P(10) to be zero blocks. The zero blocks do not really exist in the practical disk arrays. Pay attention to the fact that the selection of all-zero disks is optional, and different selections may result in a slightly different total amount of zero blocks.

disks. On the other hand, you can also add disks to the disk array easily. Adding a disk to the array can be simply treated like an update, with just two extra XOR operations per new data block.

The three optimal codes, P-Code, X-Code, and the tweaked RDP code, are all vertical RAID-6 codes. Among them, P-Code has the shortest row length in the code structure. As a practical matter, a shorter row length may perform better due to memory and caching effects [13]. Although the tweaked RDP has distributed parity across the data disks, but not like the other two codes, it still has an all-parity disk, which may become a potential bottleneck in practical applications.

7. Construction of P-Code with Prime Disks

So far, all our description about P-Code is based upon the assumption that P-Code has (prime – 1) disks in its structure. But this is not necessarily the actual case. In fact, through minimal modifications to the construction of P-Code with (prime – 1) disks, we can easily construct P-Code with (prime) disks, with all the optimal proper-ties still hold.

We have shown in section 3 that P-Code with (prime – 1) disks is based on a *partition* of the set C defined in Equation (1). Now we define a new set C^+ as follows.

$$C^+ = C \cup \{(m, n) \mid (1 \le m, n \le p-1), m \ne n, m + n = p\} \quad (10)$$

We further define p subsets of C^+, with $C^+_i = C_i$ $(1 \le i \le p-1)$, and an extra subset C^+_p is defined as in Equation (11).

$$C^+_p = \{(m, n) \mid (1 \le m, n \le p-1), m \ne n, m + n = p\} \quad (11)$$

It is easy to see that there are $(p-1)/2$ elements in subset C^+_p. This extra subset, together with the other $p-1$ subsets, form a *partition* of the set C^+. P-Code with (prime) disks is based on this *partition*.

We define P-Code with (prime) disks in a $(p-1)/2$-row-by-p-column matrix. The structure of the first $p-1$ disks is exactly the same as P-Code with (prime – 1) disks, and the last disk is assigned to be an all-data disk, with each of its $(p-1)/2$ blocks labeled with an element in C^+_p.

STEP 1. Find a prime number p, such that p is greater than n, and there is no other prime number between p and n.
STEP 2. Construct the P-Code structure with p-1 disks. i.e., label all the blocks in the structure of P-Code with p-1 disks according to Section 3.1.
STEP 3. Select p-1-n disks (the indices are $i_1, i_2, \ldots, i_{p-1-n}$) from the structure of P-Code with p-1 disks, and set all the blocks in these disks to be zero.
STEP 4. for each block (u, v) in the remaining disks, do {
if $u \in \{i_1, i_2, \ldots, i_{p-1-n}\}$ or $v \in \{i_1, i_2, \ldots, i_{p-1-n}\}$ then
set block (u, v) to be zero;
}
STEP 5. Remove all the zero blocks in the structure of P-Code with p-1 disks, and the remaining is the structure of P-Code with n disks.

Figure 9. Construction Algorithm of non-standard P-Code with n disks.

After all the blocks are labeled, the construction algorithm, proof of MDS property, and reconstruction algorithm of P-Code with (prime) disks are very similar to their counterparts of P-Code with (prime – 1) disks.

The key reason for the correctness of P-Code with (prime) disks is that the *recovery chain* for any two failed disks still exists. This is obviously right for the first $p-1$ disks, since they have exactly the same *recovery chains* as P-Code with (prime – 1) disks. If the last all-data disk is one of the two failed disks, we can still construct and prove the *recovery chain* for it and the other failed disk.

The only difference is that we use 0 as the *missing number* of the last disk. Figure 7 gives an illustrative example of this situation.

8. Construction of P-Code with Arbitrary Disks

So far, we have constructed P-Code with (prime – 1) and (prime) disks, and have also proven that they are optimal RAID-6 codes. In this section, we will construct P-Code for

an arbitrary disk array size n, where n is neither (prime) nor (prime – 1).

Pay attention to the fact that we can not construct P-Code with n disks in the same way as P-Code with (prime – 1) or (prime) disks. Since n is not prime, the *recovery chain* no longer exists for any two disks, namely, we can not traverse all the parity chains and all the blocks in the two failed disks starting from the *missing numbers* of them. Thus, the construction of P-Code with n disks must be on the basis of P-Code with (prime – 1) or (prime) disks, by assuming some disks contain all imaginary zero.

We refer to P-Code with (prime) or (prime – 1) disks as standard P-Code, and non-standard otherwise. We first construct a standard P-Code with $p - 1$ disks, where p is the smallest prime number greater than n. Next, we select (optionally) $p - 1 - n$ disks and assign all the blocks in these disks to be zero. It must be noted that we also set the parity blocks in the selected disks to be zero, so all the data blocks in the same parity chains must also be set to be zero to maintain parity consistency. Since these all-zero disks and blocks make no contribution to the disk array, we can remove them away from the structure of P-Code, and the remaining disks and blocks make up of a non-standard P-Code with n disks. Figure 8 illustrates the construction of P-Code with 8 disks, and the detailed construction algorithm for P-Code with n disks is presented in Figure 9.

Based on the correctness of the standard P-Code with (prime – 1) disks, it is easy to prove the correctness of P-Code with n disks. In particular, we can assume the imaginary all-zero disks and blocks really exist in the construction or reconstruction process, except that, since they are all zero, they need not participate in the actual XOR computation. Note that the all-zero blocks need not to be stored when deploying P-Code to disk arrays, so they do not occupy any actual storage space.

But non-standard P-Codes also have several non-optimal properties. First, each disk may not have an equal number of blocks. As we can see from Figure 8, in the structure of P-Code with 8 disks, disk d_1 has two non-zero blocks, whereas disk d_7 has three non-zero blocks. This may cause unalignment in each disk in a practical implementation. Second, each parity chain may not contain an equal number of non-zero blocks. Take the P-Code with 8 disks again for example, parity chain $P(1)$ has 7 non-zero blocks, but parity chain $P(2)$ has a different number of 6 non-zero blocks. This may increase the difficulty in buffer memory management.

For a non-standard P-Code with n disks, each disk holds one parity block and at least $(p - 3)/2 - (p - 1 - n)$ data blocks, where p is the smallest prime number greater than n. So the lower bound of the storage efficiency for a non-standard P-Code is:

$$\frac{(p-3)/2-(p-1-n)}{(p-3)/2-(p-1-n)+1} = \frac{2n-p-1}{2n-p+1} \qquad (12)$$

It must be noted that non-standard P-Codes still have the property of the flexible association between the blocks and their labels, so we can cyclically shift the labels across the disks when deploying non-standard P-Code to disk arrays. In this way, each disk in the array holds approximately the same amount of data blocks, resulting in balanced space utilization and, more importantly, balanced load per disk.

9. Conclusion

In this paper, we propose a new RAID-6 code, called the Partition Code (P-Code), for a disk array with (prime – 1) or (prime) disks. P-Code is an MDS code with optimal storage efficiency, optimal construction/reconstruction computational complexity, and optimal update complexity. P-Code complements the other two optimal RAID-6 codes, X-code and the tweaked RDP, to provide a near-full set of optimal RAID-6 configurations of typical array size (e.g., 4–20 disks).

P-code's potentially beneficial properties such as the flexible association between the labels and the data units may find useful applications in distributed storage environments., and we plan to address this problem as our future research work. P-Code's optimal properties only limited to array size (prime) or (prime – 1), and how to extend P-Code with optimal properties to different array sizes is still a challenging problem. Moreover, extending P-Code to tolerate more disk failures would also be an interesting research direction.

10. Acknowledgments

We thank the anonymous reviewers for *ICS '09* for their helpful comments. We also thank Chu Li, Bo Mao, Suzhen Wu, Lingfang Zeng, and Jianxi Chen for their feedback. This work is supported by the National Basic Research 973 Program of China under Grant No. 2004CB318201, 863 project 2008AA01A401, Changjiang innovative group of Education of China No. IRT0725, the U.S. National Science Foundation under Grant No. CCF-0621526, and the HUSTSRF No. 2007Q021B. The work of Lei Tian was done when he was working at the Computer Science & Engineering Department of the University of Nebraska–Lincoln.

11. References

[1] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *ACM SIGMOD,* ACM, pp. 109-116, June 1988.

[2] Bianca Schroeder and Garth A. Gibson. Disk Failures in the Real World: What Does an MTTF of 1,000,000 Hours Mean to You? In *FAST '07,* San Jose, CA, February 2007.

[3] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz Andre Barroso. Failure Trends in a Large Disk Drive Population. In *FAST '07,* San Jose, CA, February 2007.

[4] J. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience,* 27(9):995-1012, 1997.

[5] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An optimal scheme for tolerating double disk failure in RAID architectures. *IEEE Transactions on Computers,* 44(2):192-202, 1995.

[6] J. Hartiline. *R5X0: An efficient high distance parity-based code with optimal update complexity.* Research Report # RJ10322 (A0408-005), IBM Research Division, 2004.

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

[8] J. L. Hafner et al. *Performance Metrics For Erasure Codes in Storage Systems.* Research Report # RJ10321 (A0408-003), IBM Research Division, 2004.

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

[10] P. Corbett et al. Row-Diagonal Parity for Double Disk Failure Correction. In *Proceedings of FAST'04,* 2004.

[11] F. J. MacWilliams and N. J. A. Sloane. *The Theory of Error-Correcting Codes.* North-Holland Publishing Company, Amsterdam, New York, Oxford, 1977.

[12] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics* 8, pp. 300-304, 1960.

[13] J. Plank. The RAID-6 Liberation Codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies,* 2008.

[14] G. Zaitsev, V. Zinov'ev, and N. Semakov. Minimum-check-density codes for correcting bytes of errors, erasures, or defects. *Problems of Information Transmission,* vol. 19, pp. 197-204, 1981.

[15] L. Xu, V. Bohossian, J. Bruck, and D. Wagner. Low-density MDS codes and factors of complete graphs. *IEEE Transactions on Information Theory,* 45(6):1817-1826, 1999.

[16] Y. Cassuto and J. Bruck, Cyclic low density MDS array codes, In *Proceedings of the 2006 IEEE International Symposium on Information Theory,* pp. 2794-2798, Seattle, WA, U.S.A., July 2006.

[17] S. Nanda and N. Deo, An algorithm for a two-disk fault-tolerant array with (prime – 1) disks. *Congressus Numerantium,* vol. 171, pp. 13-23, 2004.

[18] C. Huang and L. Xu. STAR: An efficient coding scheme for correcting triple storage node failures. In *Proceedings of FAST'05,* San Francisco, December 2005.

[19] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part I: Reed-Solomon-like codes for tolerating three disk failures. *IEEE Transactions on Computers,* 54(9):1071–1080, 2005.

[20] G. Feng, R. Deng, F. Bao, and J. Shen. New efficient MDS array codes for RAID Part II: Rabin-like codes for tolerating multiple (≥4) disk failures. *IEEE Transactions on Computers*, 54(12):1473–1483, 2005.

[21] J. L. Hafner. HoVer erasure codes for disk arrays. In *Proceedings of DSN'06,* Philadelphia, June 2006.

[22] J. L. Hafner. WEAVER Codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of FAST'05,* pp. 211–224, San Francisco, December 2005.