Generalized X-Code: An Efficient RAID-6 Code for Arbitrary Size of Disk Array

XIANGHONG LUO and JIWU SHU, Tsinghua University

Many RAID-6 codes have been proposed in the literature, but each has its limitations. Horizontal code has the ability to adapt to the arbitrary size of a disk array but its high computational complexity is a major shortcoming. In contrast, the computational complexity of vertical code (e.g. X-code) often achieves the theoretical optimality, but vertical code is limited to using a prime number as the size of the disk array In this article, we propose a novel efficient RAID-6 code for arbitrary size of disk array: generalized X-code. We move the redundant elements along their calculation diagonals in X-code onto two specific disks and change two data elements into redundant elements in order to realize our new code. The generalized X-code achieves optimal encoding and updating complexity and low decoding complexity; in addition, it has the ability to adapt to arbitrary size of disk array. Furthermore, we also provide a method for generalizing horizontal code to achieve optimal encoding and updating complexity while keeping the code's original ability to adapt to arbitrary size of disk array.

Categories and Subject Descriptors: B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; H.1.1 [**Models and Principles**]: Systems and Information Theory—*Information theory*; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms: Algorithms, Performance, Reliability, Theory

Additional Key Words and Phrases: Generalized X-code, computational complexity, number of disks, size of disk array, storage

ACM Reference Format:
Luo, X. and Shu, J. 2012. Generalized X-code: An efficient RAID-6 code for arbitrary size of disk array. ACM Trans. Storage 8, 3, Article 10 (September 2012), 16 pages.
DOI = 10.1145/2339118.2339121 http://doi.acm.org/10.1145/2339118.2339121

1. INTRODUCTION

As the size of the disk array in storage systems grows larger and larger, fault tolerance becomes one of the key factors in designing new storage systems [Li et al. 2009]. Redundant Arrays of Inexpensive Disks (RAID) [Patterson et al. 1988, 1989; Chen et al. 1994] technology is widely used in modern storage systems to achieve large capacity and high reliability.

Traditional RAID technology has levels from 0 to 5 [Patterson et al. 1988], but each level tolerates at most one disk failure. As disk capacity increases much faster than bandwidth, the time needed to rebuild an entire disk is becoming longer and longer. Meanwhile, the probability of disk failures and latent sector errors arises along

This work is supported by the National Natural Science Foundation of China (Grant No. 60925006), the National Grand Fundamental Research 863 Program of China (Grant No. 2009AA01A403) and Intel International Cooperation Program (Intel-CRC-2010-06).
Author's addresses: X. Luo and J. Shu, Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China; email: luo-xh09@mails.tsinghua.edu.cn; shujw@tsinghua.edu.cn. Corresponding author: Jiwu Shu, shujw@tsinghua.edu.cn.

DOI 10.1145/2339118.2339121 http://doi.acm.org/10.1145/2339118.2339121

with the growth in size and complexity of storage systems [Bairavasundaram et al. 2007; Pinheiro et al. 2007; Schroeder and Gibson 2007]. Thus, the probability of two concurrent disk failures is increasing. For these reasons, we need RAID architectures to provide a fault tolerance of two or even more disks. RAID-6 is the first level of RAID architecture to provide a fault tolerance of two [Chen et al. 1994].

The erasure code used on RAID-6 in the literature can be divided into two categories: horizontal code and vertical code. Horizontal code indicates the class of code where redundant elements within a stripe are stored separately from the data elements, such as in EVENODD code [Blaum et al. 1995], RDP code [Corbett et al. 2004], and EEO code [Feng et al. 2010]. The other type of RAID-6 code is called vertical code, where data elements and redundant elements can be allocated through all disks in a stripe; examples are X-code [Xu and Bruck 1999], B-code [Xu et al. 1999], P-code [Jin et al. 2009], and WEAVER code [Hafner 2005].

In RAID architecture, when data elements are updated, all the redundant elements related to them must be modified as well, increasing the response time for the application. In large storage systems, intensive inputs/outputs (I/Os) are provided; data elements are frequently updated, so the complexities of encoding and updating become important metrics for measuring erasure code. A given code system defines the number of redundant elements needed to be modified with a single data element's update as the updating complexity for this element. RAID-6 code, with a fault tolerance of two, has a theoretical optimal updating complexity of two.

Different sizes of disk array are required in different storage systems. The match of disks from logic to practice is another issue to be considered in designing new erasure code. In this article, the number of disks means the logical amount of disks in theoretical analysis, while the size of disk array means the actual number of disks in real storage systems. Researchers usually use the "number of disks" in the logical design of new erasure code, but specific erasure code does not always have the ability to adapt to the arbitrary size of the disk array in real storage systems.

Of all the RAID-6 codes, horizontal code has the best the ability to adapt to the arbitrary size of the disk array, but its high computational complexity is a major shortcoming. On the other hand, vertical code's computational complexity often achieves the theoretical optimality, but vertical code is hampered by the strict requirements for the size of the disk array, which needs to be a prime number. Thus, there is a need for a novel RAID-6 code to combine the advantages of horizontal and vertical codes in order to obtain low computational complexity for the arbitrary size of the disk array.

In this article, we provide a novel efficient RAID-6 code for arbitrary size of disk array, termed "generalized X-code." We move the redundant elements along their calculation diagonals in X-code onto two specific disks and change two data elements into redundant elements to realize our new code. The new code achieves optimal encoding and updating complexity as well as low decoding complexity; in addition, it has the ability to adapt to arbitrary size of the disk array. What's more, we provide a method for generalizing horizontal code to achieve optimal encoding and updating complexity while keeping its original ability to adapt to arbitrary size of disk array.

The rest of this article is organized as follows. We will review related works in the next section and provide our motivation in Section 3. In Section 4, we will introduce our generalized X-code, including its encoding algorithm and a complete decoding analysis. In addition, we will discuss the generalized X-code's performance and properties in Section 5. A series of generalized horizontal code will also be given in Section 5. Finally, we will present our conclusions and directions for future work in the last section.

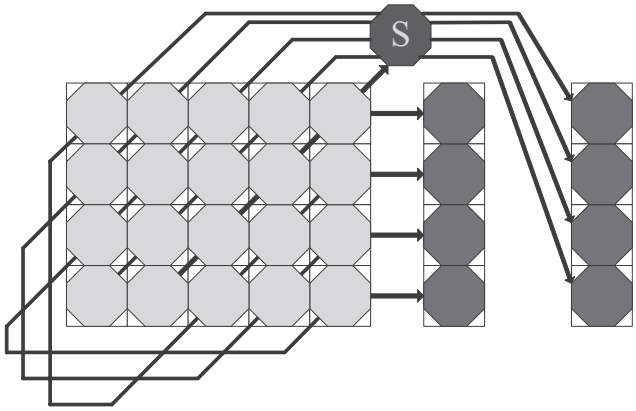


Fig. 1. EVENODD code.

2. RELATED WORK

In the study of RAID architectures and algorithms, nearly all researchers use the term "stripe" to represent a maximal set of data and redundant elements that are dependently linked by redundancy relations. "This is synonymous with 'algorithm instance' in that it is a complete instantiation of an erasure algorithm and is independent of any other instantiation" [Hafner et al. 2004]. In our article, we use the same description, discussing the RAID architecture and its corresponding algorithm in one stripe; however, in practice, disk arrays may contain many stripes.

In addition, all the redundant elements may be logically stored in the same disks, but in practice, the redundant elements can be rotated from stripe to stripe, avoiding bottlenecks when repeated write operations are performed. In the following approach of this article, all the RAID architectures will use the same rotation method from stripe to stripe.

Many RAID-6 codes have been proposed in the literature, but to date, there is no code considered to be the standard for RAID-6 because each code has its limitations. According to the distribution of data and redundant elements, RAID-6 code can be divided into two categories: horizontal code and vertical code.

2.1. Horizontal Code

Horizontal code is a class of code in which the data elements and the redundant elements are stored on different disks within a stripe; examples are EVENODD code [Blaum et al. 1995], RDP code [Corbett et al. 2004], and EEO code [Feng et al. 2010]. EVENODD code is regarded as the grandfather of horizontal code; we use it as the introductory example.

As shown in Figure 1, EVENODD specifies a storage system with $p + 2$ disks, where p should logically be a prime number. In this code, the first p disks are data disks while the last two are redundant disks. In coding theory, an element represents a chunk of data or redundant information, which is the basic building block in RAID architecture. Furthermore, a column represents a disk in the figure, so a disk failure means a column of elements cannot be accessed. In EVENODD, each disk contains $p - 1$ elements in a stripe.

As shown in Figure 1, the redundant elements in the first redundant disk are calculated by the XOR (exclusive-OR) sum of all the data elements in its row. Before producing the second redundant disk, the intermediate value, S, needs to be calculated, which represents the XOR sum of the data elements on the rightmost diagonal

(the diagonal that contains the top-right element) of slope -1 on the data disk array. The value of each redundant element in the second redundant disk is then equal to the XOR sum of S and the data elements on its corresponding diagonal of slope -1. For a clearer depiction of this process, see Figure 1.

The number of disks, p, in EVENODD, must be a prime number logically to guarantee recoverability in an arbitrary two-disk failure situation. However, because of the construction of horizontal code, data elements and redundant elements are stored on different disks, meaning the data disks do not contain any redundant elements. We can fill some of the data disks with all zeros logically, then drop these disks into a real storage system. This will not affect the rest of the code since there are no redundant elements on these disks. Under this method, the horizontal code could get rid of the limitation of requiring a prime number as the size of the disk array. For an arbitrary size of disk array, n, is needed; we can find the smallest prime number p that is not smaller than n; then logically set the number of disks as p and fill $p-n$ data disks with zero elements. By ignoring these disks in a real storage system, we can create a code that has the ability to adapt to the size of disk array, n. A disk array of size n can be regarded as shortened from a disk array of size p, so we mark this method the shorten method. Thus, horizontal code has the ability to adapt to an arbitrary size of disk array in real storage systems.

Due to the special use of the intermediate value S in the calculation of the second disk in EVENODD, when we update any data element on the calculation diagonal of S, we should first modify its corresponding redundant element on the first redundant disk, then modify all the redundant elements on the second redundant disk, so the updating complexity of the data elements on the diagonal of S is p, while the updating complexity of other data elements is two. So the average updating complexity in EVENODD code is larger than two.

Unfortunately, Blaum et al. [1996] and Blaum and Roth [1999] prove that standard horizontal code cannot attain an updating complexity of two. Liberation code [Plank 2008] combines with Liber8tion code [Plank 2009] and Blaum-Roth code [Blaum and Roth 1999] to form a class of RAID-6 codes called the "minimal density codes," achieving an optimal updating complexity for horizontal code, but it is still larger than two. Clearly, computational complexity is the most obvious shortcoming of horizontal code.

2.2. Vertical Code

In vertical code, redundant elements are not necessarily separated from data elements. Redundant elements and data elements within a stripe can be stored on any disk; examples are X-code [Xu and Bruck 1999], B-code [Xu et al. 1999], P-code [Jin et al. 2009], and WEAVER code [Hafner 2005]. Among them, X-code [Xu and Bruck 1999] is the most popular. We will use X-code as an example to discuss the properties of vertical code.

As is shown in Figure 2, X-code contains p disks, and each disk contains p elements, where p must be a prime number. On each disk, the first $p-2$ elements are data elements while the last two are redundant elements.

Figure 2(a) illustrates the calculation relationship for the first redundant row. Each redundant element in this row is calculated by the XOR sum of all the data elements on its diagonal of slope -1. Figure 2(b) shows that each redundant element in the second redundant row equals the XOR sum of all the data elements on its upper elements' diagonal of slope 1.

Each data element in X-code is protected by exactly two redundant elements, and all of the redundant elements depend only on data elements, but not on each other. As a result, the average updating complexity of X-code is exactly two (when we update a

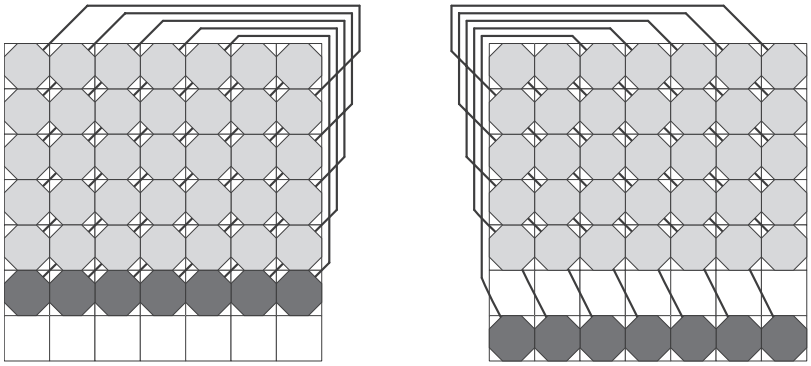


Fig. 2. X-code.

single data element in X-code, we need to modify its corresponding redundant element in the last two rows, each row with exactly one redundant element). This is the theoretically optimal updating complexity for erasure code with a fault tolerance of two. Moreover, the encoding and decoding complexity of X-code also attain the theoretical optimality.

The number of disks, p, in X-code should be a prime number, and other vertical codes in the literature all have similar limitations. However, the redundant elements in vertical code are spread across all the disks and they need to be consistent with their calculation diagonals; thus they cannot be set as zeros artificially. As a result, we cannot use the same "shorten" method as in horizontal code to make vertical code adapt to an arbitrary size of disk array. Furthermore, vertical code is difficult to extend to a higher fault tolerance code, since the redundant elements are allocated throughout all the disks.

3. MOTIVATION

In this section, we will compare the two categories of RAID-6 code and point out the properties that our novel code needs to provide.

In mathematics, erasure code with a fault tolerance of two has a theoretical optimal updating complexity of two. Many vertical codes could attain this bound, but Blaum et al. [1996] and Blaum and Roth [1999] have proved that horizontal code cannot attain it. Since intensive I/Os are provided in large storage systems, data elements are frequently updated; the lack of encoding and updating efficiencies creates a bottleneck, preventing the implementation of horizontal code.

The size of disk array in vertical code must be a prime number, which may potentially waste space. Though horizontal code also has the limitation of requiring a prime number of disks logically, we can get rid of this limitation in practice by the "shorten" method to make it adapt to arbitrary size of disk array.

In addition, we can easily add disks to a horizontal code system. Since each data disk contains only data elements, adding a data disk to the code system can be simply regarded as an update for each data element on this disk. But for vertical code, adding new disks requires changing all the diagonals, which means all the redundant elements need to be recalculated.

Both categories of RAID-6 code have their own advantages and disadvantages. Computational complexity is the main shortcoming for horizontal code, but it has the ability to adapt to arbitrary size of disk array. In contrast, vertical code limits the size of disk

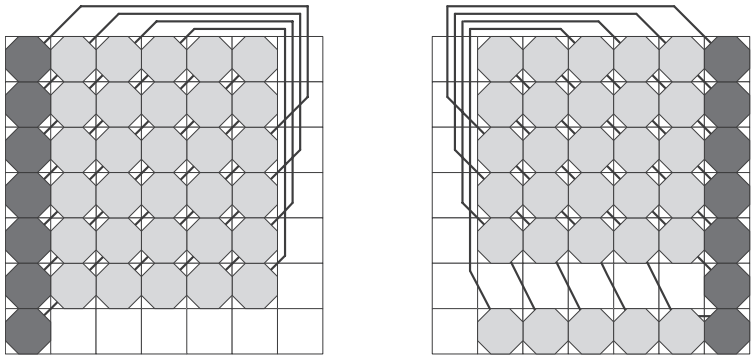


Fig. 3. Code construction after moving the redundant elements along their calculation diagonals in X-code.

array to be a prime number, but its computational complexity often achieves theoretical optimality. Is there a code that could combine their advantages, that is, having both low computational complexity and the ability to adapt to arbitrary size of disk array? This is the motivation of our novel work.

4. GENERALIZED X-CODE

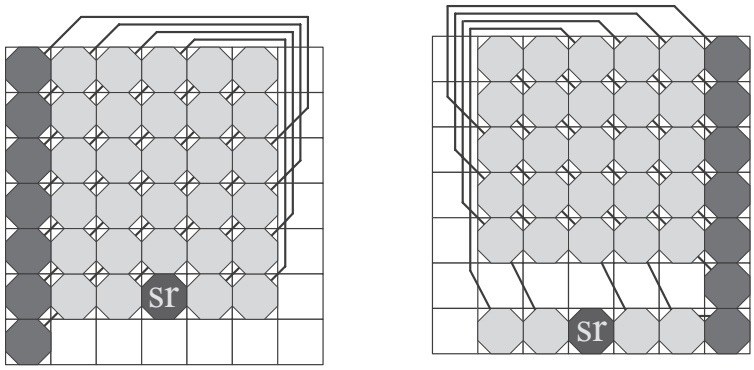
X-code achieves optimal encoding and updating complexity, so one method to realize our proposed code is to change the arrangement of elements in X-code into horizontal code form, while keeping the calculation diagonals unchanged, to ensure that encoding and updating complexity do not rise.

In traditional horizontal code, redundant elements are allocated to two columns (two disks), but in X-code, the redundant elements are allocated to the last two rows (across all the disks). So we need to move the redundant elements in X-code onto two of the columns to realize our idea. To keep the original calculation relationships, we just move the redundant elements along their calculation diagonals. After these moves, the new structure is as shown in Figure 3.

As shown in Figure 3(a), we move the redundant elements that were originally in the first redundant row into the first column, to constitute the first redundant disk. Specifically, the second element originally in the first redundant row is moved to the left-down position since it is on the extension cords of its calculation diagonal of slope -1.

In a similar fashion, we move the redundant elements originally in the second redundant row into the last column to constitute the second redundant disk in Figure 3(b). We further move the originally last redundant element in the last column up to the penult position and create a new redundant element in the last position of the second redundant disk, which records the XOR sum of the elements on the calculation diagonal of the previous penult element in the second redundant row in X-code.

In these movements, we do not break the basic calculation diagonals, keeping the data elements in the first $p - 2$ rows with an updating complexity of two. However, a new problem arises: the data elements in the last two rows only relate to one redundant element, which means the failures of one of these data elements and its corresponding redundant element will lead to data loss. That is to say, after these movements, the code only provides a fault tolerance of one, but what we need is an erasure code that could tolerate failures on any two disks.



(a) (b)

Fig. 4. Generalized X-code.

4.1. Encoding

Our novel code is based on the previously illustrated movements, but we make slight modifications to the resulting code to bring it to a fault tolerance of two.

According to our analysis, each data element in the last two rows is related to only one redundant element, so in order to enable our code to provide a fault tolerance of two, we need other redundant elements to protect all these data elements. Unfortunately, classic horizontal code cannot achieve an updating complexity of two, as is proven by Blaum et al. [1996] and Blaum and Roth [1999], so adding new redundant elements or changing the calculation equations of some redundant elements on the redundant disks is useless. We must make some changes on the data disks.

An example of our novel code is proposed in Figure 4, where a column represents a disk. As the elements marked with "sr" are shown in Figure 4, we replace a data element with a redundant element in each of the last two rows. The value of this redundant element is equal to the XOR sum of all the data elements in its row. We call these two elements the "special-redundant" (sr) elements. In addition, since our novel erasure code is generalized from X-code, we call it "generalized X-code."

We describe the systematic construction of generalized X-code in mathematical language in the following.

We assume there are p disks in generalized X-code, where the 0^{th} disk and the $(p-1)^{th}$ disk are redundant disks containing only redundant elements. The other $p-2$ disks, indexed from 1 to $p-2$, are data disks containing data elements and two "special-redundant" elements. In addition, each disk contains p elements. More importantly, p should be a prime number logically. This requirement is very important to ensure recoverability in arbitrary two-disk failures. However, we could get rid of this limitation in real storage systems, as will be discussed in Section 5.5.

Definition. Consider a $p \times p$ disk array, where p should be a prime number, we define the element $a_{i,j}$ $(0 \leq i < p, 0 \leq j < p)$ as the j^{th} element on the i^{th} disk in the disk array.

In the following description, we use the standard mathematical notation $\langle x \rangle_y$ to represent the remainder when x is divided by y. That is to say, $\langle x \rangle_y = z$ if and only if $x \equiv z \pmod{y}$ $(0 \leq z < y)$. For example, $\langle 5 \rangle_3 = 2$ and $\langle -1 \rangle_5 = 4$. In addition, we use the mathematical notation $\bigoplus$ to represent the XOR operation.

In order to conveniently describe the calculation formulas of redundant elements in the first redundant disk and the special-redundant element in the penult row, we

use $b_{i,j}$ $(0 < i < p-1, 0 \le j < p)$ to represent the j^{th} data element on the i^{th} disks in Figure 4(a). The special-redundant element in the penult row will be regarded as zero in the array $\{b_{i,j}\}$, and all the elements in the last row of data disks will also be regarded as zeros in the array $\{b_{i,j}\}$. That is,

$$\begin{gathered} b_{i,j} = a_{i,j} \quad (0 < i < p-1, 0 \le j < p-2), \\ b_{i,p-2} = a_{i,p-2} \quad (0 < i < p-1, i \ne \tfrac{p-1}{2}), \\ b_{\frac{p-1}{2},p-2} = 0, \\ b_{i,p-1} = 0 \quad (0 < i < p-1). \end{gathered}$$

With $\{b_{i,j}\}$, the calculation formula for the first redundant disk can be expressed as follows.

$$a_{0,j} = \bigoplus_{i=1}^{p-2} b_{i,\langle j-i\rangle_p} \quad (0 \le j < p).$$

The special-redundant in the penult row can be expressed as

$$a_{\frac{p-1}{2},p-2} = \bigoplus_{i=1}^{p-2} b_{i,p-2}.$$

In a similar fashion, in order to conveniently describe the calculation formulas of redundant elements in the second redundant disk and the special-redundant element in the last row, we use $c_{i,j}$ $(0 < i < p-1, 0 \le j < p)$ to represent the j^{th} data element on the i^{th} disks in Figure 4(b). The special-redundant element in the last row and all the elements in the penult row of data disks will be regarded as zeros, finally, the last two rows will exchange with each other in array $\{c_{i,j}\}$. That is,

$$\begin{gathered} c_{i,j} = a_{i,j} \quad (0 < i < p-1, 0 \le j < p-2), \\ c_{i,p-2} = a_{i,p-1} \quad (0 < i < p-1, i \ne \tfrac{p-1}{2}), \\ c_{\frac{p-1}{2},p-2} = 0, \\ c_{i,p-1} = 0 \quad (0 < i < p-1). \end{gathered}$$

With $\{c_{i,j}\}$, the formula for the second redundant disk can be expressed as follows.

$$a_{p-1,j} = \bigoplus_{i=1}^{p-2} c_{i,\langle j+i+1\rangle_p} \quad (0 \le j < p).$$

In addition, the special-redundant in the last row can be expressed as

$$a_{\frac{p-1}{2},p-1} = \bigoplus_{i=1}^{p-2} c_{i,p-2}.$$

4.2. Decoding

The generalized X-code defined in the preceding provides a fault tolerance of two, which means this code could recover all the failed elements from arbitrary two-disk failures. In this section, we will enumerate all of the possible failure situations and provide the corresponding decoding strategies to verify this property.

Without loss of generality, we assume the disks i and j ($0 \leq i < j < p$) fail. We have three different failure situations to enumerate and verify their recoverability.

$i = 0$ and $j = p - 1$. This means the two failed disks are exactly the two redundant disks; thus no data elements have been lost. We can simply use the encoding process to reconstruct these failed disks.

$i = 0$ and $0 < j < p - 1$. In this case, one redundant disk and one data disk fail. We use the other intact redundant disk to help recover the lost elements.

As is shown in Figure 4(b), we can ignore disk 0. The data elements in the first $p - 2$ rows on disk j are obviously related to different redundant elements on disk $p-1$. That is to say, each data element in the first $p-2$ rows on disk j is the only failed element in its corresponding calculation diagonal of slope 1, so we can recover it by calculating the XOR sum of all the other elements on its corresponding calculation diagonal of slope 1 in Figure 4(b). Then, as all the other elements in the last two rows are available, we can recover the last two elements in disk j by calculating the XOR sum of all the other elements in each of the last two rows thanks to the calculation formulas of the special-redundant elements, no matter they are the exact special-redundant elements or other data elements. Finally, we can recover the elements on the first redundant disk by their encoding formulas.

In the same light, this decoding strategy can also be applied to the failure situation where $0 < i < p - 1$ and $j = p - 1$.

$0 < i < j < p - 1$. This is the main situation. Both failed disks are data disks. In Figure 4(a), each data disk has at most $p - 1$ elements affecting the first redundant disk. The same is true for the second redundant disk in Figure 4(b). However, there are p diagonals in each figure. As a result, for each data disk, there is at least one diagonal that does not cross it. It is not hard to deduce that the $(x - 1)^{th}$ diagonal from the top does not cross disk x in Figure 4(a) and the $(x + 1)^{th}$ diagonal from bottom does not cross disk x in Figure 4(b).

In Figure 4(a), since the $(i - 1)^{th}$ diagonal of slope -1 does not cross disk i, there is only one data element lost on the $(i - 1)^{th}$ diagonal of -1, that is $a_{j,\langle i-j-1\rangle_p}$, so this data element can be easily recovered by the XOR sum of all the other elements on this diagonal. After this data element has been recovered, there is only one data element on its diagonal of slope 1 in Figure 4(b) left to be recovered, so we can recover $a_{i,\langle 2(i-j)-1\rangle_p}$ as well. We then return to Figure 4(a) and continue the recovery process, until the row coordinate reaches $p - 1$ or $p - 2$ (there is no element on row $p - 1$ that crosses any calculation diagonal in Figure 4(a) and no element on row $p - 2$ that crosses any calculation diagonal in Figure 4(b)). Thus, we can get a recovery chain.

Similarly, in Figure 4(a), the $(j-1)^{th}$ diagonal of slope -1 does not cross any element on the $(j-1)^{th}$ disk. And in Figure 4(b), the $(i+1)^{th}$ diagonal of slope 1 does not cross any element on the i^{th} disk, nor does the $(j+1)^{th}$ diagonal of slope 1 for the j^{th} disk. So, we can get four recovery chains, and their corresponding recovery sets can be represented as:

$$
\begin{aligned}
A_0 &= \{a_{i,\langle 2k(i-j)-1\rangle_p}, \quad a_{j,\langle (2k-1)(i-j)-1\rangle_p}, \quad k = 1, 2, \ldots\},\\
A_1 &= \{a_{i,\langle (2k-1)(j-i)-1\rangle_p}, \quad a_{j,\langle 2k(j-i)-1\rangle_p}, \quad k = 1, 2, \ldots\},\\
B_0 &= \{a_{i,\langle 2k(j-i)-1\rangle_p}, \quad a_{j,\langle (2k-1)(j-i)-1\rangle_p}, \quad k = 1, 2, \ldots\},\\
B_1 &= \{a_{i,\langle (2k-1)(i-j)-1\rangle_p}, \quad a_{j,\langle 2k(i-j)-1\rangle_p}, \quad k = 1, 2, \ldots\}.
\end{aligned}
$$

As is shown in Figure 5, we provide an example of decoding steps for better understanding of the retrieval chains. We assume disks 1 and 2 are failed in this example. The recovery chains of sets A_0 and A_1 are shown in Figure 5(a), while those of sets B_0 and B_1 are shown in Figure 5(b). As is shown in Figure 5, combining these four

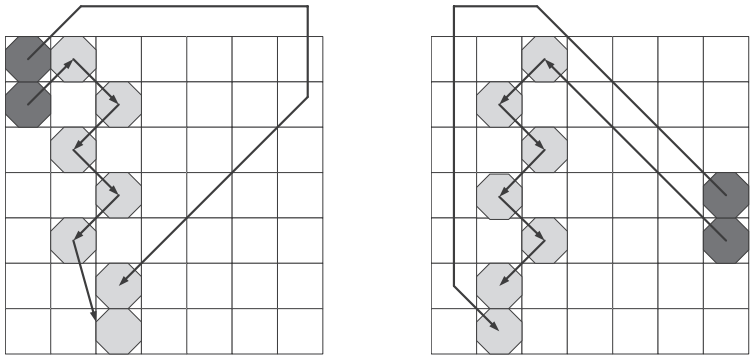


Fig. 5. An example of the recovery chains.

recovery chains, we can recover all the failed elements. In the following, we prove that the four recovery chains always cover all the failed elements for any specific i and j.

We can easily combine the four recovery sets into two sets:

$$C = \{a_{i,\langle t(j-i)-1\rangle_p}\},$$

and

$$D = \{a_{j,\langle t(j-i)-1\rangle_p}\}.$$

Because there is only one calculation diagonal that crosses each element in the last two rows, we start each recovery chain from row $p-1$, and end it at row $p-2$. Thus the domain for t in these two sets is determined by the following congruence equation, which represents the situation in which the row coordinate reaches $p-2$.

$$t(j-i)-1 \equiv p-2 \pmod{p}.$$

Because $0 < i < j < p-1$, it is easy to deduce $0 < j-i < p-1$. Since t can be either negative or positive, it can be represented as

$$t_1(j-i) \equiv 1 \pmod{p} \quad (t_1 > 0),$$

or

$$t_2(j-i) \equiv p-1 \pmod{p} \quad (t_2 > 0).$$

If we add the two congruence equations together, we will get

$$(t_1+t_2)(j-i) \equiv 0 \pmod{p}.$$

Because p is a prime number and $0 < j-i < p-1$, we can conclude that t_1+t_2 should be a multiple of p.

In either set C or D, if two of the elements are equal to each other, there exist certain q_1 and q_2 that satisfy the following congruence equation.

$$q_1(j-i)-1 \equiv q_2(j-i)-1 \pmod{p}.$$

Since $0 < j-i < p-1$, and p is a prime number, we will get

$$q_1 \equiv q_2 \pmod{p}.$$

From all of these analyses, we know that either C or D has p different elements, which are:

$$C = \{a_{i,r}, \quad 0 \leq r < p\},$$

and

$$D = \{a_{i,r}, \quad 0 \leq r < p\}.$$

From the mathematical analysis, these four recovery chains travel through all of the elements in the two failed disks, so we can recover all the data elements on disk i and disk j.

When neither disk i nor disk j contains a special-redundant element, the decoding process is finished. However, if one of them is exactly the $(\frac{p-1}{2})^{th}$ disk, we need to recover the two special-redundant elements. Since all the data elements in the last two rows are available after the previous recover process, it is an easy job to recover the two special-redundant elements via their calculation formulas.

Single-disk failure. If the failed disk is one of the redundant disks, no data is lost and we can recover it by its encoding formula. If the failed disk is one of the data disks, since there is no diagonal that crosses the same disk twice, as seen in Figure 4(a) or Figure 4(b), we can recover the first $p - 2$ elements in the failed disk by their corresponding calculation diagonals in either Figure 4(a) or Figure 4(b). Then we can use the calculation formulas for the special-redundant elements to recover the last two elements in the failed disk.

By enumerating all the failure situations, we prove that we can always find a decoding strategy for any of them. So we have proved that generalized X-code provide a fault tolerance of two.

5. PERFORMANCE AND PROPERTY ANALYSIS

5.1. Evaluation Principles

We only provide the RAID-4-like [Chen et al. 1994] architecture of our generalized X-code in a stripe in this article, but it is synonymous with the relationships between RAID-4 and RAID-5, the mappings of logical disks to practical disks are rotated from stripe to stripe in real storage systems.

Multiple stacks are implemented in real storage systems; each stack contains all different possible mappings from logical disks to practical disks, which means the loss of any two physical disks covers all combinations of failures of two logical disks in one stripe. We can logically regard all the disks in RAID-6 as having the same opportunity to be updated on their elements and the same possibility of failing. With these assumptions, we can do the measurements in the following evaluations by rigorous counting and averaging on a single stripe [Hafner et al. 2004].

In the following figures, all the numbers are generated by rigorous mathematics derivations; we use the same size of data disk array to compare generalized X-code with other RAID-6 codes to ensure the fairness of comparisons in the performance metrics.

We have different mappings from logical disks to practical disks in a stack, so the two redundant logical disks in a stripe will be mapped to different practical disks with equal possibilities. This rotation of the redundant elements in practice in RAID-6 naturally avoids bottleneck effects when repeated write operations are performed; either in the encoding or the updating process. Each disk has equal possibility that it contains the elements to be modified and the corresponding redundant elements to be updated, so we can naturally get load-balance.

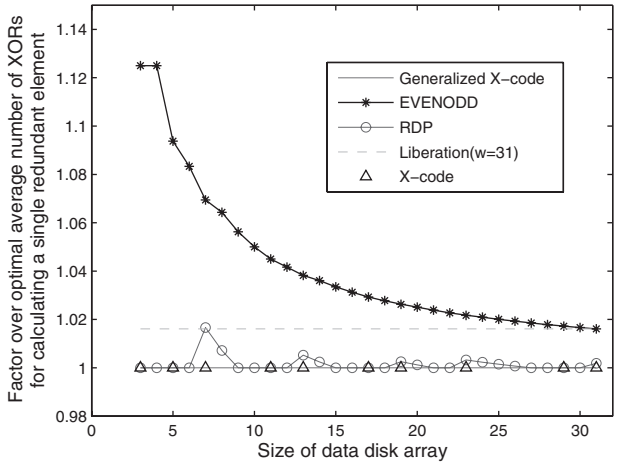


Fig. 6. Comparison of encoding complexity for RAID-6 codes.

5.2. Encoding Complexity

We use the average number of XORs in calculating a single redundant element as the metric to measure the encoding complexity. This includes both the elements on the redundant disks and the special-redundant elements.

Since the optimal number of XORs can be easily generated from a specific number of data elements and redundant elements in RAID-6, we can normalize the results by dividing the average number of XORs by its corresponding optimal number of XORs to achieve a better metric for encoding complexity comparison. Obviously, through this normalization, the optimal value becomes one. The factors over the optimal average number of XORs for calculating a single redundant element are illustrated in Figure 6 to compare the encoding complexity of RAID-6 codes.

Generalized X-code achieves theoreticaly optimal encoding complexity for arbitrary size of data disk array, as is shown in Figure 6. It is because each data element in generalized X-code is related to exactly two redundant elements, and the redundant elements depend only on data elements but not on each other. It is easy to see that, besides generalized X-code, no other horizontal RAID-6 code achieves consistent optimal encoding complexity in all possible sizes of data disk array, which is also proved by Blaum et al. [1996] and Blaum and Roth [1999]. X-code, by contrast, achieves optimal encoding complexity only in some specific size of disk array.

In large storage systems, intensive I/Os are required. So the encoding complexity greatly affects the response time in large writes while updating complexity is the key factor for the response time in small writes.

5.3. Updating Complexity

We measure the updating complexity as the average number of redundant elements that need to be modified with the update of a single data element. The updating complexities of different RAID-6 codes are illustrated in Figure 7 for comparison.

As is shown in Figure 7, the updating complexity in either EVENODD code [Blaum et al. 1995] or RDP code [Corbett et al. 2004] increases with the size of disk array, reaching its upper-bound at three. Liberation code, the optimal horizontal code [Plank 2008], greatly improves the updating complexity of RAID-6 horizontal code but it still cannot reach two.

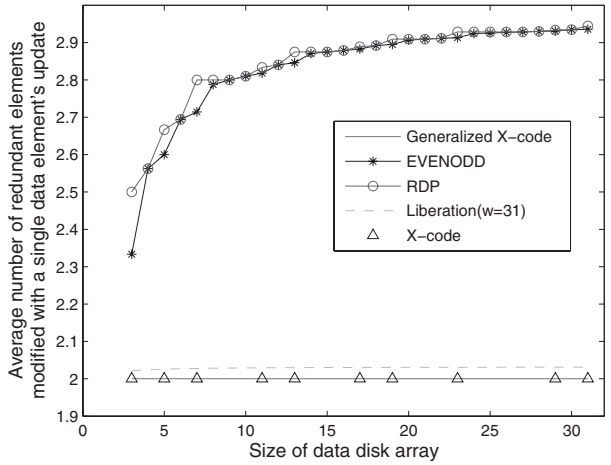


Fig. 7. Comparison of updating complexity for RAID-6 codes.

Each data element in the first $p-2$ rows in generalized X-code affects one redundant element in the first redundant disk and one redundant element in the second redundant disk. In addition, each data element in the penult row affects one redundant element in the first redundant disk and the special-redundant element in the same row (i.e.: $a_{\frac{p-1}{2},p-2}$). Similarly, each data element in the last row is related to one redundant element in the second redundant disk and the special-redundant element in this row (i.e.: $a_{\frac{p-1}{2},p-2}$). Moreover, all the redundant elements depend only on data elements, but not on each other. All in all, the update of any single data element in generalized X-code only requires two redundant elements to be modified; thus, the updating complexity of generalized X-code is two, which achieves the theoretical optimality.

5.4. Decoding Complexity

In this section, we enumerate all the possible combinations of two disk failures to analyze the decoding complexity. We use the average number of XORs needed to recover a single element as the metric to evaluate the decoding complexity. As with encoding, we can normalize the result by dividing it by its corresponding optimal number of XORs to obtain a clearer metric for decoding complexity comparison. The factors over the optimal average number of XORs needed to recover a single element are illustrated in Figure 8 to compare the decoding complexity for RAID-6 codes.

In general, RDP [Corbett et al. 2004] exhibits the best decoding complexity, and generalized X-code performs better than EVENODD [Blaum et al. 1995]. In addition, generalized X-code exhibits better decoding performance as the size of data disk array increases, approaching optimality.

Based on the characteristics of generalized X-code, different shorten strategies affect the decoding complexity. Since the special-redundant elements are not protected by the redundant disks, the diagonals that cross the special-redundant elements contain fewer elements than other diagonals; there are fewer XOR operations in the decoding process if we use these diagonals. As a result, the best strategy is to fill up all zeros to the data disks that are close to the middle disk (the disk that contains the special-redundant elements), and drop them in practice.

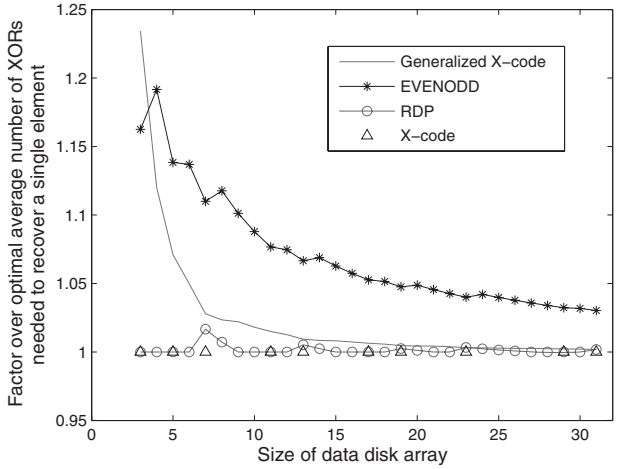


Fig. 8. Comparison of decoding complexity for RAID-6 codes.

5.5. Ability to Adapt to Arbitrary Size of Disk Array

Vertical code also exhibits optimal performance on encoding, updating, and decoding complexities, but generalized X-code provides an advantage over the limitation of requiring a prime number size of disk array, which creates a bottleneck in vertical code.

In traditional vertical codes, redundant elements are spread across all the disks, so code systems cannot drop any disk. As a result, the size of disk array in vertical code must be a prime number to ensure recoverability in arbitrary two-disk failures.

In generalized X-code, we should keep the number of disks as a prime number logically to ensure the recoverability in arbitrary two-disk failures. But it is quite different from vertical code; the redundant elements in generalized X-code are only allocated on three disks. Therefore, we can logically set some of the data disks to contain data elements with only imaginary zeros, and drop them in real storage systems. This will not affect the rest of the code system since there are no redundant elements on these zero-disks. With this shorten method, generalized X-code has the ability to get rid of the limitation of requiring a prime number as the size of the disk array. For any disk array size, n, we can find the smallest prime number p that is not smaller than n, then logically set the number of disks as p, and fill $p - n$ data disks with all zeros. By ignoring these disks in a real storage system, we can create a code system of size n. So the generalized X-code has the ability to adapt to arbitrary size of disk array.

More specifically, we can not only conveniently shorten the size of disk array but also easily add disks to the disk array. Adding a disk to the system can be regarded as putting an all-zero data disk into use, where we can treat this addition as an update to all the data elements on this disk, with just two extra XOR operations per element (because each data element is related to exactly two redundant elements in generalized X-code). This has a great advantage over vertical code, where adding new disks changes all the diagonals, requiring recalculation of all the redundant elements.

5.6. Generalize Horizontal Code to Attain an Updating Complexity of Two

Generalized X-code is generalized from original vertical code to combine the advantages of both horizontal and vertical code. Conversely, a natural question would be whether we can modify some original horizontal codes to achieve the same effects.

The reason that EVENODD code [Blaum et al. 1995] cannot get an updating complexity of two is the use of S. When updating the data elements on the diagonal of S, we

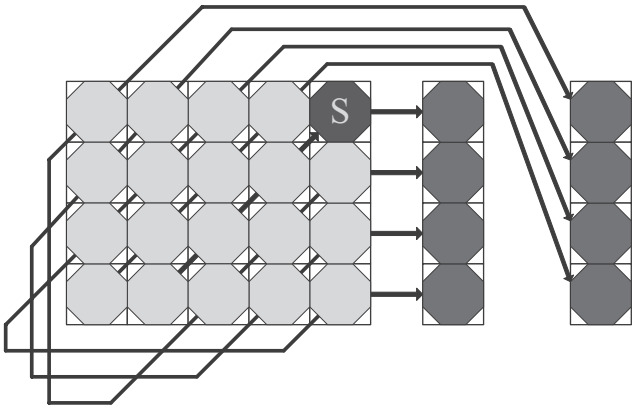


Fig. 9. Generalized EVENODD code.

need to modify all the redundant elements in the second redundant disk, thus increasing the average updating complexity. However, if we replace one of the data elements on the diagonal of S with a redundant element, the situation will change. As shown in Figure 9, we change the top-right data element into a redundant element to record the XOR sum of other data elements on its diagonal of slope -1. In addition, we do not include this element for the calculation of the first element in the first redundant disk. Under this change, the value of each redundant element on the second redundant disk is exactly equal to the XOR sum of the data elements on its corresponding diagonal of slope -1, no longer including S.

With these modifications, the new generalized EVENODD code achieves optimal encoding and updating complexity and keeps its original ability to adapt to arbitrary size of disk array as well. The generalized EVENODD code's ability to provide a fault tolerance of two can be easily verified. This method of generalizing horizontal code to achieve optimal encoding and updating complexity and simultaneously keep its original ability to adapt to arbitrary size of disk array as well, can be used on most of the horizontal codes, such as RDP code [Corbett et al. 2004].

6. CONCLUSIONS AND FUTURE WORK

In this article, we have proposed a novel efficient RAID-6 code, termed generalized X-code, for arbitrary size of disk array. We move the redundant elements along their calculation diagonals in X-code to keep their original calculation relationships and change two data elements into redundant elements to ensure that the code provides a fault tolerance of two. With these operations, we obtain our novel RAID-6 code, generalized X-code. The new code achieves optimal encoding and updating complexity and low decoding complexity, as well as the ability to adapt to arbitrary size of disk array. We have further proposed a method to generalize horizontal code to achieve optimal encoding and updating complexity while keeping its original ability to adapt to arbitrary size of disk array.

The main deficiency of the generalized X-code is its storage efficiency. Generalized X-code is not strict MDS (maximum distance separable) code, but since we have only changed one (in generalized EVENODD code) or two (in generalized X-code code) data elements into redundant elements from MDS code, there is little impact on storage efficiency compared with MDS code. It has been previously proven that MDS horizontal code cannot attain an updating complexity of two [Blaum et al. 1996; Blaum and Roth 1999]; however, it has not yet been proven whether MDS vertical code could break the

limitation that the size of disk array must be a prime number. Therefore, future work should focus on the possibility of an MDS vertical code that could both obtain the ability to adapt to arbitrary size of disk array and retain optimal computational complexity.

ACKNOWLEDGMENTS

The authors would like to thank MingQiang Li and Letian Yi for discussing our ideas, YingQian Liu and Em Turner Chitty for their kind help with the English language, and the editors and reviewers for their constructive comments.

REFERENCES

BAIRAVASUNDARAM, L. N., GOODSON, G. R., PASUPATHY, S., AND SCHINDLER, J. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. ACM, New York, 289–300.

BLAUM, M., BRADY, J., BRUCK, J., AND MENON, J. 1995. EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. *IEEE Trans. Comput. 44*, 2, 192–202.

BLAUM, M., BRUCK, J., AND VARDY, A. 1996. MDS array codes with independent parity elements. *IEEE Trans. Inf. Theory 42*, 2, 529–542.

BLAUM, M. AND ROTH, R. M. 1999. On lowest density MDS codes. *IEEE Trans. Inf. Theory 45,* 1, 46–59.

CHEN, P. M., LEE, E. K., GIBSON, G. A., KATZ, R. H., AND PATTERSON, D. A. 1994. RAID: High-performance, reliable secondary storage. *ACM Comput. Surv. 26,* 2, 145–185.

CORBETT, P., ENGLISH, B., GOEL, A., GRCANAC, T., KLEIMAN, S., LEONG, J., AND SANKAR, S. 2004. Row-diagonal redundant for double disk failure correction. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 2–15.

FENG, J., CHEN, Y., AND SUMMERVILLE, D. 2010. EEO: An efficient MDS-like RAID-6 code for parallel implementation. In *Proceedings of the 33rd IEEE Sarnoff Symposium*. IEEE, Piscataway, NJ, 1–5.

HAFNER, J. L. 2005. WEAVER codes: Highly fault tolerant erasure codes for storage systems. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 211–224.

HAFNER, J. M., DEENADHAYALAN, V., KANUNGO, T., AND RAO, K. 2004. Performance metrics for erasure codes in storage systems. Tech. rep. RJ 10321, IBM Research, San Jose, CA.

JIN, C., JIANG, H., FENG, D., AND TIAN, L. 2009. P-code: A new RAID-6 code with optimal properties. In *Proceedings of the 23rd International Conference on Supercomputing (ICS)*. ACM, New York, 360–369.

LI, M., SHU, J., AND ZHENG, W. 2009. GRID codes: Strip-based erasure codes with high fault tolerance for storage systems. *ACM Trans. Storage 4*, 4, 1–22.

PATTERSON, D. A., CHEN, P., GIBSON, G., AND KATZ, R. H. 1989. Introduction to redundant arrays of inexpensive disks. In *Proceedings of the IEEE COMPCON Conference*. IEEE, Los Alamitos, CA, 112–117.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, 109–116.

PINHEIRO, E., WEBER, W. D., AND BARROSO, L. A. 2007. Failure trends in a large disk drive population. In *Proceedings of 5th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 17–28.

PLANK, J. 2008. The RAID-6 liberation codes. In *Proceedings of 6th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 97–110.

PLANK, J. 2009. The RAID-6 liber8tion code. *Internat. J. High Perform. Comput. Appl. 23*, 3, 242–251.

SCHROEDER, B. AND GIBSON, G. A. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*. USENIX, Berkeley, CA, 1–16.

XU, L., BOHOSSIAN, V., BRUCK, J., AND WAGNER, D. G. 1999. Low-density MDS codes and factors of complete graphs. *IEEE Trans. Inf. Theory 45*, 6, 1817–1826.

XU, L. AND BRUCK, J. 1999. X-code: MDS array codes with optimal encoding. *IEEE Trans. Inf. Theory 45,* 1, 272–276.

Received April 2011; revised November 2011, March 2012; accepted May 2012