Two-Erasure Codes from 3-Plexes

Liping Yi, Rebecca J. Stones, Gang Wang

Two-Erasure Codes from 3-Plexes

Liping Yi, Rebecca J. Stones, and Gang Wang

College of Computer Science, Nankai University, Tianjin, China
{yiliping,becky,wgzwp}@nbjl.nankai.edu.cn

Abstract. We present a family of parity array codes called 3-PLEX for tolerating two disk failures in storage systems. It only uses exclusive-or operations to compute parity symbols. We give two data/parity layouts for 3-PLEX: (a) When the number of disks in array is at most 6, we use a horizontal layout which is similar to EVENODD codes, (b) otherwise we choose hybrid layout like HoVer codes. The major advantage of 3-PLEX is that it has optimal encoding/decoding/updating complexity in theory and the number of disks in a 3-PLEX disk array is less constrained than other array codes, which enables greater parameter flexibility for trade-offs in storage efficiency and performances.

Keywords: Latin squares · array codes · storage efficiency · computational complexity · data/parity layout.

1 Introduction

As a fault-tolerant technology, erasure codes have been widely used to ensure the reliability of storage systems. According to whether encoding/decoding is based on exclusive-or operations, erasure codes can be roughly divided into two categories: Reed-Solomon codes and parity array codes. The common property of these codes is that they tolerate two simultaneous disk failures, but they also have their own trade-offs in terms of storage efficiency and computational complexity.

Traditional Reed-Solomon code [12] is based on finite field operations which results in high computational complexity. Quite a few studies have tried to improve the computational performance of RS codes by using special hardware or dedicated algorithms. For example, the modified Cauchy RS code [13] replaces the finite field operation with exclusive-or operation, but it still has higher computational complexity than array codes. Although RS codes have the above shortcoming, it is the only erasure code that can achieve arbitrary fault tolerance, and it is an MDS code [2] which means it has optimal storage efficiency. Another advantage of RS codes is that there is no assumption on the number of disks which enables it to be applied in more scenarios.

Parity array codes arrange data/parity blocks according to the structure of arrays [5]. Compared with RS codes, since array codes are completely based on exclusive-or operations, its encoding and decoding algorithms are relatively simple and easy to implement. According to the data/parity layout, array codes

can be divided into three types: (a) horizontal codes (such as EVENODD [1], RDP [3], etc.), (b) vertical codes (X-Code [16], WEAVER [8]), etc.), and (c) hybrid codes (HoVer [9]). All of these array codes tolerate two faults.

EVENODD code [1] as the original horizontal array code, provides an efficient means to tolerate double disk failures in RAID architecture. Fig. 1-a shows an example of EVENODD with five data disks and two parity disks. EVENODD code only uses two individual parity disks (stores horizontal parities and secondary diagonal parities respectively) to achieve two fault tolerance and the implementation of EVENODD code does not require any hardware modification of the standard RAID-5 controller [1]. It's also a MDS code with optimal storage efficiency. Another advantage of EVENODD is that the number of disks of EVENODD disk array is adjustable. The array size of EVENODD is $(n-1) \times (n+2)$, where n is the number of data disks. Although its encoding definition requires prime n, it can also achieve arbitrary number of disks by horizontal shortening. Since EVENODD code defines a special check sum S (the XOR-sum of "$D4$"s in Fig. 1-a), its computational complexity is higher than other array codes.

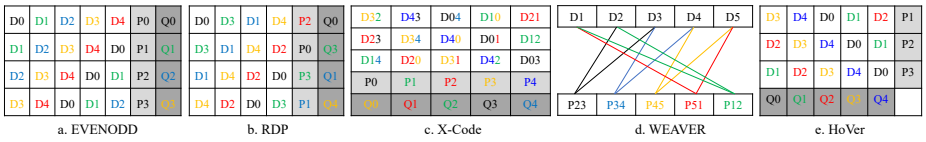


Fig. 1. Array codes of 5 data blocks

RDP (Row-Diagonal Parity) code [3] is another horizontal array code whose array size is $(p-1) \times (p+1)$ (where p is a controlling parameter which must be a prime number greater than 2). RDP code replaces the special check sum S of EVENODD with a missing diagonal and uses a "parity dependent" architecture (where row parities also contribute to the calculation of diagonal parities), which reduces the computational complexity compared with EVENODD. Fig. 1-b illustrates a RDP disk array with four data disks and two parity disks. Corbett et al. [3] proved that both encoding/decoding and updating complexity of RDP is almost optimal.

Updating complexity is an important metric of array codes; we generally use the number of blocks required to be updated when a data block is updated to measure updating complexity. In a storage system with frequent updating operations, higher updating complexity will lead to worse performance. In order to alleviate this problem, vertical array codes have been proposed.

X-Code [16] is the first vertical array code which tolerates two disk failures. The array size of X-Code is $n \times n$ (where n is a prime); the first $n-2$ rows are filled with data blocks, and the last two rows store parity blocks, i.e., each disk stores both data and parity blocks. X-Code computes the XOR sum of all data blocks on the diagonals of slope 1 and -1 to generate parity blocks, hence the

name X-Code. Fig. 1-c depicts an example of X-Code with 5 disks. This special structure enables X-Code to achieve optimal computational complexity. X-Code is also an MDS code, and thus it has optimal storage efficiency. An important limitation of X-Code is that the number of disks in X-Code array must be a prime and it cannot admit arbitrary array sizes using horizontal shortening like EVENODD. If we impose simple horizontal shortening on a X-Code to break through its constraint of array size, the original relationship between data blocks and corresponding parity blocks will be destroyed, thereby losing the ability to tolerate two erasures.

WEAVER code [8] is another typical vertical array code. There are four types of WEAVER codes proposed in [8], and only WEAVER(n, k, t) when $k = t = 2$ can tolerate two disk failures. The array size of WEAVER$(n, 2, 2)$ is $2 \times n$, the first row only stores data blocks and the last row is filled with parity blocks. Fig. 1-d shows an example of WEAVER$(n, 2, 2)$ with five disks. WEAVER code computes parity blocks by calculating XOR sum of two adjacent data blocks like weaving, hence named WEAVER. The definition of encoding enables WEAVER to have no constraint in array size, which makes up for the deficiency of X-Code. Besides, a key advantage of WEAVER is that it can achieve up to twelve fault tolerance by adjusting parameters. It also has optimal computational complexity. Nonetheless, the main disadvantage of WEAVER code is that it has low storage efficiency (up to 50%). In other words, WEAVER code has a trade-off among storage efficiency, fault tolerance and computational complexity.

Hybrid array codes combine the advantages of horizontal and vertical array codes. HoVer code [9] with $(v + r) \times (h + n)$ array size is a representative hybrid array code, hence named HoVer (Horizontal and Vertical). Fig. 1-e shows an example of HoVer code with 5×8 array size. The advantages of HoVer code is that it's a near-MDS code, which means it has high but not optimal storage efficiency, no limitation of array size, parameter flexibility enables it to achieve higher fault tolerance, and its updating complexity is optimal. However, the encoding and decoding complexity become higher as the number of disks increases, so HoVer code also has a trade-off between storage efficiency and encoding/decoding speed.

Hafner [8] pointed out that there is no perfect array code, and each code has a trade-off among fault tolerance, storage efficiency and computational complexity. In this paper, we aim to find an array code with a better trade-off.

We follow ideas in e.g. [4–7] describing methods of using Latin squares to construct array codes. Motivated by these methods, this paper proposes a new double-erasure array code named 3-PLEX based on Latin squares. The advantages of 3-PLEX are that it has optimal computational complexity and no constraint of array size. The storage efficiency of 3-PLEX is always 60%. In other words, 3-PLEX also has a trade-off between storage efficiency and performances.

This paper is organized as follows: we give the relationship between Latin squares and k-plex in Section 2. Then, we describe the encoding procedure of 3-PLEX with mathematical definition in Section 3. In Section 4, we present the decoding procedure with proof for two fault tolerance. In Section 5, we compare 3-PLEX and other array codes. In Section 6, we describe an implementation

and performance tests of 3-PLEX based on NCFS. Section 7 summarizes the correspondence and presents some future works.

2 Latin squares and k-plexes

A k-plex of order n is an $n \times n$ matrix whose symbols belong to a set of size n, with the following properties:

- each row contains k distinct symbols and $n - k$ empty positions,
- each column contains k distinct symbols and $n - k$ empty positions, and
- each symbol occurs exactly k times.

Latin squares are k-plexes of order n when $k = n$, and thus k-plexes generalize Latin squares [14]. A 1-plex embedded in a Latin square is called a transversal. The union of 3 disjoint transversals forms a 3-plex, and we use this method to construct 3-plexes. Fig. 2-a is a Latin square of order 5 containing the 3-plex of order 5 in Fig. 2-b. Fig. 2-c indicates to obtain a 3-PLEX code.

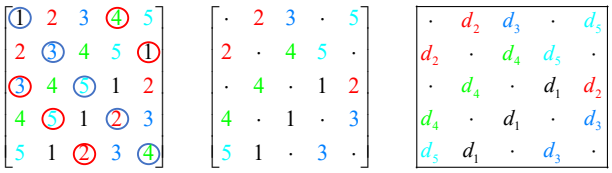


Fig. 2. The relationship among DCLS, 3-plex and 3-PLEX code

In this paper, we construct 3-plexes from diagonally cyclic Latin squares (DCLSs), which we define as a Latin square having forward broken diagonals which have a constant increment from top to bottom (this is slightly different from the standard definition [15]). Each broken diagonal in the Latin square in Fig. 2-a increases by 2 (we highlight two forward broken diagonals). In a DCLS, the union of any three forward broken diagonals is a 3-plex. For simplicity, we use back-circulant Latin squares (i.e., constant broken antidiagonals) as DCLSs. DCLSs only exist when n is odd, although 3-plexes exist for all orders $n \geq 3$.

3 Encoding

Assuming that there are n data disks, in Section 3.1 we discuss how to arrange the parity blocks. We also assume each block occupied the space of 1 bit. In actual storage, each block may occupy 1 byte or more of disk space, which will not affect the encoding procedure of 3-PLEX. Similar to other array codes [1,3,9,16], the parity blocks in 3-PLEX code can also be computed by calculating the XOR

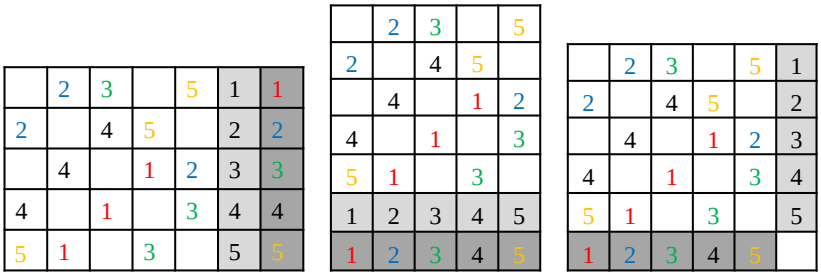
sum of data blocks on the diagonal with slope 1 or -1. Fig. 2 shows how to map a 3-plex of order 5 to a 3-PLEX code with five data disks. Data blocks in a 3-PLEX array correspond to symbols in the 3-plex marked with the same colors. The data blocks placed in the same column are stored in the same disk in the 3-PLEX disk array. Note that data blocks are placed consecutively on each disk rather than arranged in line with the data layout of 3-PLEX code, i.e., the empty positions denoted '.' do not actually exist. To guarantee 3-PLEX with two-fault tolerance, we encode data blocks in 3-PLEX array to row parity blocks and diagonal parity blocks respectively like EVENODD code.

3.1 Data/parity layout

Before giving the definition of the encoding procedure, we need to choose one appropriate data/parity layout of 3-PLEX. There are three layouts introduced previously: horizontal, vertical, and hybrid layout. The storage efficiency and encoding/updating complexity of the three layouts of 3-PLEX are same, and only decoding complexity will be meaningfully affected by data/parity layout, therefore in order to attain the lowest decoding complexity, we need to adapt a proper data/parity layout.

Let's take a 3-PLEX code of order 5 as a running example. Fig. 3 plots three data/parity layouts of 3-PLEX. The row parity blocks are shaded light gray, and the diagonal parity blocks are shaded dark gray and represented by the same number as their data blocks. In real storage systems, typically blocks in each layout compose a *stripe*, and blocks in the same column form a *stripe unit* as the basic storage unit. Stripe units in a stripe are stored on different disks. A storage system contains many stripes. In addition, the empty positions in each array layout do not exist actually and blocks in the same column are consecutively stored practically. Thus, data blocks and parity blocks have different sizes in horizontal layout and hybrid layout. This will not cause unbalance in storage because data and parity blocks from different stripes typically are evenly distributed over all the disks in a modern distributed storage system. Moreover, encoding/decoding/updating computation will be performed correctly as long as we keep track of the relationship between data blocks and parity blocks.

We compute the average number of exclusive-or operations required to reconstruct two failed disks to measure the decoding complexity of 3-PLEX with different data/parity layouts separately. For horizontal layout, we calculate the decoding complexity from threes cases: (a) when two data disks fail, the decoding complexity is 12, (b) when one data disk and one parity disk fail: $6 + 2n$ (where n is the number of data disks), and (c) when two parity disks fail: $4n$. Hence, the mean decoding complexity of reconstructing two disk failure is $6+2n$. In terms of vertical layout, we find it can not support two fault tolerance after theoretical analysis. As for hybrid layout, we compute the decoding complexity in two ways: (a) two hybrid disks fail: 14, and (b) one hybrid disk and row parity disk fail: $24+2n$. Thus, the horizontal layout has lower decoding complexity than the hybrid layout when $n < 6$, otherwise the hybrid layout has lower decoding complexity. For simplicity, we focus on 3-PLEX with horizontal layout.



a. Horizontal layout b. Vertical layout c. Hybrid layout

Fig. 3. Three alternative data/parity layouts of 3-PLEX

3.2 Encoding procedure

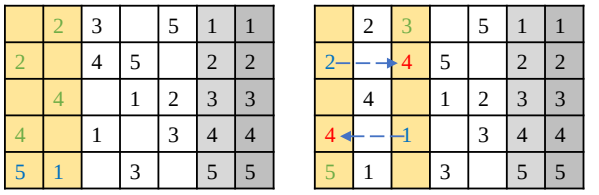
Let $a_{i,j}$ be the data/parity block at the ith row and jth column ($i \in \{0, 1, \ldots, n-1\}$ and $j \in \{0, 1, \ldots, n+1\}$), then the parity blocks of 3-PLEX code with horizontal layout can be calculated according to the following rules:

$$a_{i,n} = \bigoplus_{j=0}^{n-1} a_{i,j}, \tag{1}$$

$$a_{i,n+1} = \bigoplus_{j=0}^{n-1} a_{\langle i-j \rangle_n, j}. \tag{2}$$

where n is the number of data disks and $\langle x \rangle_n = x \bmod n$. For simplicity, we choose a 3-plex from three non-consecutive forward diagonals so that the difference between the indices of two of the three is indivisible by 3.

In order to make the encoding rules defined by (1) and (2) easy to understand, we give an encoding example of 3-PLEX with horizontal layout and order five showed in Fig. 4-a.



a. Two consecutive failed data disks b. Two inconsecutive failed data disks

Fig. 4. Two sub-cases of two failed data disks

To illustrate the encoding rules of 3-PLEX by taking the first row parity block $a_{0,5}$ and the first diagonal parity block $a_{0,6}$ illustrated in Fig. 4-a as specific

examples:

$$a_{0,5} = a_{0,1} \oplus a_{0,2} \oplus a_{0,4}$$
$$a_{0,6} = a_{4,1} \oplus a_{3,2} \oplus a_{2,2}$$

Other row or diagonal parity blocks can be calculated with same manner. In addition, it's easy to see that the number of data blocks in each row parity group and diagonal parity group is same, i.e., the number of exclusive-or operations required to compute one row parity block and one diagonal parity block is equal. This feature explains why the encoding/updating complexity of horizontal and hybrid layouts are identical.

4 Decoding and proof for two fault tolerance

When a single disk of failure occurs in 3-PLEX, lost data blocks of failed disk can be directly reconstructed through row parity or diagonal parity. In this Section, a corresponding decoding algorithm is given to prove that 3-PLEX code has the ability to correct two disk failures.

Assuming columns c_1 (disk c_1) and c_2 (disk c_2) fail in the 3-PLEX array, we describe the decoding algorithm split into four cases:

Case: $c_1 = n$; $c_2 = n+1$. I.e., both of the parity disks fail. This case is equivalent to re-encoding all data blocks, hence the decoding algorithm is the same as the encoding procedure.

Case: $c_1 < n$; $c_2 = n$. I.e., one data disk and the row parity disk fail simultaneously. In this case, the invalid data disk should be first recovered through diagonal parity blocks and surviving data blocks, and then the failed row parity disk can be reconstructed according to the encoding rules of row parity. Hence, the decoding algorithm can be derived on the basis of (1) and (2):

$$a_{i,c_1} = (\bigoplus_{j=0}^{n-1} a_{<i-j>_n,j}) \oplus a_{i,n+1}, \quad j \neq c_1,$$
$$a_{i,n} = a_{i,c_2} = \bigoplus_{j=0}^{n-1} a_{i,j}.$$

Case: $c_1 < n$; $c_2 = n+1$. I.e., one data disk and the diagonal parity disk fail simultaneously. This situation is similar to the previous case. we can directly reconstruct failed data disk using row parity blocks and survival data blocks, then failed diagonal disk can be recovered according to (2). Therefore, the decoding algorithm of this case can be derived from (1) and (2):

$$a_{i,c_1} = (\bigoplus_{j=0}^{n-1} a_{i,j}) \oplus a_{i,n}, \quad j \neq c_1,$$
$$a_{i,n+1} = a_{i,c_2} = \bigoplus_{j=0}^{n-1} a_{<i-j>_n,j}.$$

Case: $c_1 < c_2 < n$. I.e., two data disks fail simultaneously. We divide this case into two sub-cases which are illustrated in Fig. 4: the two failed data disks are (a) consecutive, (b) not consecutive. For case (a), since any consecutive two data disks only have at most one common row filled with data blocks, we recover the two data blocks in this special row (if it exists) according to diagonal parity; we mark these two data blocks blue in Fig. 4. The remaining data blocks are only one failed data blocks in their respective rows, and are thus simply recovered via row parity, marked in green in Fig. 4. As for case (b), we recover the failed data blocks marked green on the basis of row parity, and we also recover data blocks marked blue according to diagonal parity. After the blue data blocks are recovered, then we use row parity blocks and other related survival data blocks to recover the last unrepaired data blocks marked red. Hence, the decoding algorithm of situation (b) can be derived according to (1) and (2):

$$a_{i,c_1} = \left(\bigoplus_{j=0}^{n-1} a_{<i-j>_n,j}\right) \oplus a_{i,n+1}, \quad j \neq c_1,$$

$$a_{i,c_2} = \left(\bigoplus_{j=0}^{n-1} a_{i,j}\right) \oplus a_{i,c_1} \oplus a_{i,n}.$$

Since the decoding algorithms above are all derived from encoding rules defined by (1) and (2), we conclude that 3-PLEX code has two fault tolerance.

5 Comparison with existing schemes

In this section, we compare 3-PLEX code with array codes proposed in [1,3,8,9,16] in terms of fault tolerance, storage efficiency, encoding/decoding/updating complexity and constraint of array size.

Fault tolerance: fault tolerance is a basic indicator for measuring erasure codes. Tab. 1 lists the fault tolerance and requirements for the number of data disks of multiple erasure codes. RS code is the only code applied in practice

Table 1. Comparison of multiple erasure codes in fault tolerance

Erasure code	Fault tolerance	The number of data disks
RS	Arbitrary	Depends on the length of code words
EVENODD	2	Prime
RDP	2	Prime−1
X-Code	2	Prime
WEAVER	≤ 12	No constraints
HoVer	Arbitrary	No constraints
3-PLEX	2	No constraints

that can support arbitrary fault tolerance, but it has higher computational complexity. Although HoVer code also can achieve arbitrary fault tolerance, its high

fault tolerance will significantly influence storage efficiency and computational complexity. EVENODD, RDP and X-Code can only tolerant two simultaneous disks failures, and they also have array size limitation. WEAVER code can support up to 12 fault tolerance due to its special definition, however there is no systematic construction method for WEAVER codes and most of them need to determine the encoding rules using special search technology, hence WEAVER code has worse scalability. 3-PLEX code proposed in this paper has been proved that it has two fault tolerance.

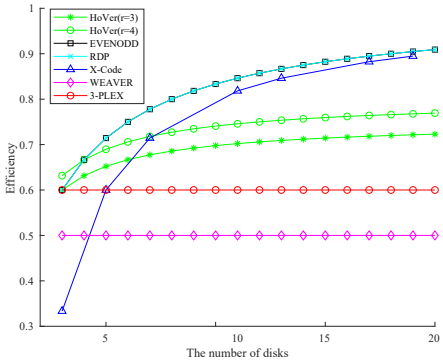


Fig. 5. Storage efficiency

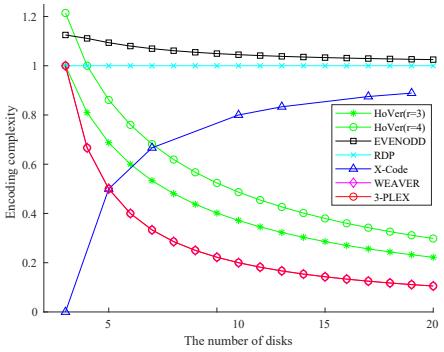


Fig. 6. Encoding complexity

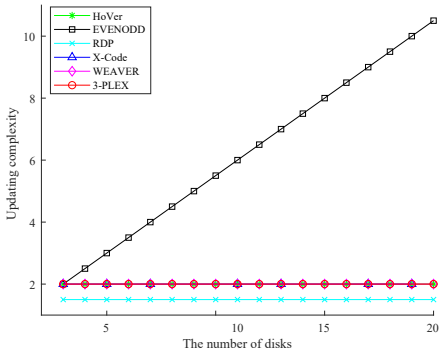


Fig. 7. Updating complexity

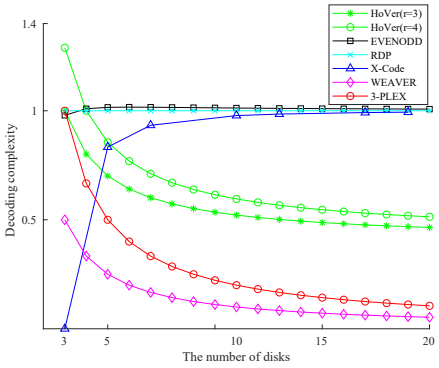


Fig. 8. Decoding complexity

Storage efficiency: Storage efficiency is the proportion of data blocks among all blocks, and is another important criterion for evaluating erasure codes. In Fig. 5, we see that WEAVER$(n, 2, 2)$ code has the lowest storage efficiency (50%) which is a trade-off between storage efficiency with other indicators. The storage efficiency of 3-PLEX code is always 60% which is higher than WEAVER$(n, 2, 2)$ code. Since EVENODD, RDP, and X-Code are MDS codes, and HoVer is a near-

MDS code, they have (near) optimal storage efficiency. Therefore, 3-PLEX also has a trade-off between storage efficiency and the other performance metrics.

Encoding complexity: Encoding overhead of erasure codes determines practicability, and hence is an important indicator for evaluation. We generally use the number of exclusive-or operations required to generate one single parity block during encoding procedure to measure the encoding complexity. Fig. 6 plots the encoding complexity vs. the number of disks. In order to facilitate the observation of the relationship between practical encoding complexity and theoretical optimal encoding complexity (the generation of a single parity block requires $n-1$ exclusive-or operations in theory), we use the ratio of practical and theoretical optimal encoding complexity as the ordinate, i.e., integer 1 in ordinate indicates that the encoding complexity is theoretically optimal. It can be seen from Fig. 6 that the encoding complexity of WEAVER code and 3-PLEX code is equal and lowest compared to other array codes. HoVer, X-Code and RDP codes reach the optimal encoding complexity in theory. Since the encoding procedure of EVENODD code involves a special checksum S, its encoding complexity approaches but is not theoretical optimal as the number of disks increases.

Updating complexity: A large number of small IO operations increases the updating overhead of a storage system, hence updating complexity is an important parameter for judging the performance of erasure codes. Updating complexity is usually represented by the average number of parity blocks needing recalculation after updating one data block. Fig. 7 illustrates the variation in updating complexity of the various array codes as the number of data disks increases. Except for EVENODD code, the updating complexity of other array codes is 2 which implies these array codes have optimal updating complexity. Since EVENODD requires the special checksum S due to the calculation of diagonal parity blocks, when one data block on this special diagonal is updated, S also will be updated and all diagonal parity blocks need to be updated, this is the reason why EVENODD's updating complexity is higher than other array codes.

Decoding complexity: The decoding complexity of erasure codes directly affects the availability and reliability of a storage system. Therefore, decoding complexity is another key indicator for evaluating erasure codes. We usually use the number of exclusive-or operations required to recover a single failed block during decoding procedure to measure decoding complexity. We use the ratio of practical and theoretical optimal decoding complexity as ordinate of Fig. 8 and integer 1 indicates that the decoding complexity of corresponding array code is optimal in theory. We see in Fig. 8 that 3-PLEX, WEAVER$(n, 2, 2)$, X-Code and RDP code all reach or approach the theoretical optimal decoding complexity. Also due to the existence of S, EVENODD's decoding complexity is higher than other array codes but also close to be optimal in theory.

To summarize, 3-PLEX code exchanges lower computational complexity with some storage space, i.e., there is a trade-off between storage efficiency and the other performance metrics.

6 Implementation and Performance

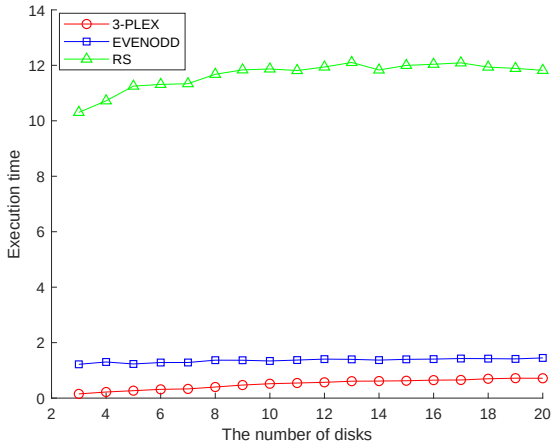


Fig. 9. Single fault reconstruction time of 3-PLEX, EVENODD and RS code

In this section, we implement the encoding and decoding algorithm of 3-PLEX code on Network-Coding-Based Distributed File System (NCFS) [10]. We focus on measuring the decoding complexity of repairing after single disk failure of each erasure code on NCFS [11]. We deploy NCFS on the virtualized Linux platform with 8 cores (2GHz) of CPU, 16G of memory, 200G of HDD, and use actual execution time of reconstructing one failed disk to represent the decoding complexity of each erasure codes. For each erasure code, we respectively measure its execution time of decoding 100 times when the total number of disks in this code ranges from 3 to 20 and calculate the average of the results of 100 times to record the decoding time. Fig. 9 plots the test results for RS, EVENODD and 3-PLEX code.

We observe that 3-PLEX code has the lowest decoding complexity, EVENODD code follows, and RS code has the highest decoding complexity. As we assume that the capacity of each disk is 100MB, it seems that 3-PLEX code has a marginal improvement over EVENODD code illustrated in Fig. 9. If each disk stores far greater than 100MB of data, the advantage of 3-PLEX code will be more obvious. Another conclusion we draw is that the decoding complexity of the three erasure codes becomes gradually stable as the number of disks increases to 15. Both of the two conclusions are consistent with the performance analysis in theory. In addition, we observe that performances of each code in the simulated storage system are in agreement with the theoretical analysis in Section 5, which indicates that the theoretical performance comparison analysis of the codes in Section 5 is reliable.

7 Conclusions

In this paper, we present a novel erasure code called 3-PLEX code which can tolerate two disk failures. It has optimal computational complexity and no constrains of array size. It exchanges low computational complexity with storage efficiency, i.e., there is a trade-off between storage efficiency and performances. In addition, since the idea for 3-PLEX code generated from Latin squares, we can continue to research the following aspects: (a) study 3-plex and orthogonal 3-plexes to extend 3-PLEX code with higher fault tolerance, (b) adjust the values of k and n to find a k-PLEX code that can achieve a better trade-off in fault tolerance, storage efficiency and performances.

References

1. Blaum, M., Brady, J., Bruck, J., Menon, J.: EVENODD: An efficient scheme for tolerating double disk failures in RAID architectures. IEEE Trans. Comput. **44**(2), 192–202 (1995)
2. Blaum, M., Roth, R.M.: On lowest density MDS codes. Trans. Inform. Theory **45**(1), 46–59 (1999)
3. Corbett, P., English, B., Goel, A., Grcanac, T., Kleiman, S., Leong, J., Sankar, S.: Row-diagonal parity for double disk failure correction. In: Proc. FAST. pp. 1–14 (2004)
4. Gang, W., Sheng, L., Xiaoguang, L., Jing, L.: Representing X-Code using latin squares. In: Proc. PRDC. pp. 177–182 (2009)
5. Gang, W., Xiaoguang, L., Sheng, L., Guangjun, X., Jing, L.: Constructing double-erasure HoVer codes using Latin squares. In: Proc. ICPADS. pp. 533–540 (2008)
6. Gang, W., Xiaoguang, L., Sheng, L., Guangjun, X., Jing, L.: Constructing liberation codes using latin squares. In: Proc. PRDC. pp. 73–80 (2008)
7. Gang, W., Xiaoguang, L., Sheng, L., Guangjun, X., Jing, L.: Generalizing RDP codes using the combinatorial method. In: Proc. NCA. pp. 93–100 (2008)
8. Hafner, J.L.: WEAVER codes: Highly fault tolerant erasure codes for storage systems. In: Proc. FAST. vol. 5 (2005)
9. Hafner, J.L.: HoVer erasure codes for disk arrays. In: Proc. DSN. pp. 217–226 (2006)
10. Hu, Y., Yu, C.M., Li, Y.K., Lee, P.P., Lui, J.C.: NCFS: On the practicality and extensibility of a network-coding-based distributed file system. In: Proc. NetCod. pp. 1–6. IEEE (2011)
11. Huang, C., Xu, L.: Star: An efficient coding scheme for correcting triple storage node failures. Trans. Comput. **57**(7), 889–901 (2008)
12. MacWilliams, F.J., Sloane, N.J.A.: The theory of error-correcting codes, vol. 16. North Holland (1977)
13. Plank, J.S.: Optimizing Cauchy Reed-Solomon codes for fault-tolerant storage applications. University of Tennessee, Tech. Rep. CS-05-569 (2005)
14. Wanless, I.M.: A generalisation of transversals for Latin squares. Electron. J. Combin (2002), r12
15. Wanless, I.M.: Diagonally cyclic Latin squares. Euro. J. Combin **25**, 393–413 (2004)
16. Xu, L., Bruck, J.: X-Code: MDS array codes with optimal encoding. IEEE Trans. Inform. Theory **45**, 272–276 (1999)